Assignment 3

Overview

A program such as the game made in Assignment 2 can be easy to develop at first, but later can be difficult to modify. For example, not only would the `World` module have to be entirely rewritten if a height coordinate was to be added, but anything that used the module would have to be modified as well. The `Item` module has similar limitations, although these are reduced by the use of an encapsulated record. There is also the danger that a well-meaning programmer will modify the internal values of these modules and, without realizing, change them to a state the other software is not prepared to cope with.

> **Refactor** [definition]: To rewrite existing source code in order to improve its readability, reusability, or structure without affecting its meaning or behaviour.

The accepted way to reduce this problem is to refactor the modules into encapsulated classes. This prevents the internal values from being modified unexpectedly, and allows the internal architecture to be modified with minimal or no effect on the rest of the program. In this assignment, you will refactor the `World` and `Item` modules to encapsulated classes and move the description array into the `World` class. You will also adapt the class interfaces to be more flexible by replacing the row and column coordinates with a single `Location` value. This change in the interface will significantly increase the flexibility of the modules by allowing greater freedom in their implementations.

Another problem is that the descriptions are stored in a global variable (the description array). This means that, if the program were to load two different `World`s, the descriptions for the second world would overwrite those from the first. We will fix this by moving the description array into the `World` class. We will also make the array much larger and use a member field to store the number of elements currently stored in the array. With this change, we will be able to use the same program to load worlds with different numbers of descriptions.

The purpose of this assignment is to ensure that you understand how to use simple C++ classes including constructors and member functions, and to show you how classes can aid in data encapsulation. For Part A, you will create a type to represent a location in the world. For Part B, you will refactor your existing `World` module from Assignment 1 into a class. For Part C, you will refactor your existing `Item` module from Assignment 2 into a class. For part D, you will incorporate these classes in your game program and load a new game world.

Copy the code and data files of your Assignment 2. Do not just modify Assignment 2.

(If you are using Visual Studio, you must start by creating a new project for Assignment 3. Do NOT copy the whole folder including the `.sln` file or massive confusion will result!)

Requirements

Part A: The `Location` Type [15% test program]

Create a class named `Location` to represent a location in the world. The `Location` type should have its own interface (`.h`) and implementation (`.cpp`) files. The `Location` class will not be encapsulated, but only the `World` class will look at its internal fields.

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

- `Location ();`
- `Location (int row1, int column1);`
- `bool operator== (const Location& other) const;`

An ***operator*** is a special function that has a valid C++ symbol as part of its name. Once you have defined `operator==`, you can use `==` in a normal manner to compare two `Location` variables.

The `Location` type will also have an associated non-member function with the following prototype:

- ```
  ostream& operator<< (ostream& out,
                       const Location& location);
  ```

The above operator should be declared in `Location.h` but its prototype should be placed outside of the `Location` class. Likewise, the implementation of the operator should be placed in `Location.cpp` but it should not have "`Location::`" in front of the operator name.

Perform the following steps:

1. Define `Location` as a class with `public` `row` and `column` fields. These are both `int`s. Since the data fields are not `private`, the `Location` class is not encapsulated.
    - **Note:** The test program requires the exact names `row` and `column`.
2. Add a default constructor to the `Location` type that sets the row to `0` and the column to `0`.
3. Add an initializing constructor to the `Location` type that takes a row and a column as parameters. The constructor should initialize the corresponding member fields in the `Location` to these values.
    - **Note:** Every constructor for a class must set the value of every member variable in the class.
4. Add a constant equality test operator (`operator==`) for `Location`s. It should take a constant reference to another `Location` as a parameter and return `true` if the row of this instance is equal to the row of the other instance and the column of this instance is equal to the column of the other instance.
5. Add a stream output operator (`operator<<`) for `Location`s as a non-member function in the same file. It should take a non-constant reference to an `ostream` (the type of `cout`) and a constant reference to a `Location` as parameters. Then it should print the row and column of
  ```

the `Location` to the `ostream` (using << syntax) and return a non-constant reference to the `ostream`. Print the row and column in the following format:
`(row = 2, column = 4)`
Do not print a newline.

- **Hint:** Look at online notes in Section 07-2 of the course: http://www2.cs.uregina.ca/~anima/115/Notes/07-Overloading/OperatorOverloading.html
- **Note:** You must match this format exactly. If you do not, the test program will dock you marks.

6. Compile and link your `Location` module with the `TestLocation3.cpp` program provided. You will also need the `TestHelper.h` and `TestHelper.cpp` files. Run the resulting program. It should give you full marks.
   - **Hint:** `g++ Location.cpp TestHelper.cpp TestLocation3.cpp –o run`

Part B: Refactor `World` into a Class [35% = 15% test program + 10% code + 10% documentation]

Refactor the `World` type to be an encapsulated class, and change it to use `Location`s instead of row and column pairs. The 2D node array and the 1D description array will become member variables. The functions associated with the `World` type will become member functions.

By the end of Part B, your `World` type will have `public` member functions with the following prototypes:

```
World (const string& game_name);
void debugPrint () const;
bool isValid (const Location& location) const;
bool isDeath   (const Location& location) const;
bool isVictory (const Location& location) const;
bool canGoNorth (const Location& location) const;
bool canGoSouth (const Location& location) const;
bool canGoEast  (const Location& location) const;
bool canGoWest  (const Location& location) const;
Location getNorth (const Location& location) const;  // new in A3
Location getSouth (const Location& location) const;  // new in A3
Location getEast  (const Location& location) const;  // new in A3
Location getWest  (const Location& location) const;  // new in A3
Location getStart () const;
void printStartMessage () const;
void printEndMessage () const;
void printDescription (const Location& location) const;
```

Note: There is no default constructor.

Note: The `worldClear` function has been removed.

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

- **`void loadNodes (const string& filename);`**
- **`void loadDescriptions (const string& filename);`**
- **`bool isInvariantTrue () const;  // new in A3`**

Perform the following steps:

1. Change the `ROW_COUNT` and `COLUMN_COUNT` constants to both have a value of `10`. This is the size of the Ghostwood world, which the test programs for this assignment assume you are using.

    - **Note:** After you change these constants, your game (the `main` function) will not run again until Part D.

2. Replace the `DESCRIPTION_COUNT` constant in `World.h` with a new constant named `MAX_DESCRIPTION_COUNT`. It should have a value of `1000`.

3. Replace the `World typedef` with a `World class` that has `private` fields for `nodes`, `descriptions`, and `description_count`. The `nodes` field is a 2D array of `ROW_COUNT` by `COLUMN_COUNT NodeValue` elements, and takes the place of the old `World typedef`. The `descriptions` field is a 1D array of `MAX_DESCRIPTION_COUNT strings`, similar to the old global array. The `description_count` field is an `unsigned int` and stores how many elements of the `descriptions` array are being used.

4. Delete the `worldClear` function.

5. Convert the `worldLoadNodes` function to a `private` helper function named `loadNodes`. It should load the nodes values into the `nodes` array.

6. Convert the `worldLoadDescriptions` function to a `private` helper function named `loadDescriptions`. It should store the number of descriptions in the `description_count` variable. Then it should load that many descriptions into the `descriptions` array. Assume that the description count in the file will never be larger than `MAX_DESCRIPTION_COUNT`.

    - **Hint:** If you have an existing variable named `description_count` in the `worldLoadDescriptions` function, you should remove its declaration when you convert this function into the `loadDescriptions` function, because you want the number of descriptions to be stored in the `description_count` field of the `World` instance rather than in a local variable.

7. Convert the `worldLoadAll` function into a constructor with one parameter, the game name. It should call the `loadNodes` function to initialize the `nodes` array and it should also call the `loadDescriptions` function to initialize the `descriptions` and `description_count` fields. Thus, every member variable will be initialized.

8. Convert the `worldDebugPrint` function to a constant member function named `debugPrint`. First it should print the values in the `nodes` array in a grid. Then it should print the description count and then the descriptions themselves, separated by blank lines.

9. Convert the `worldIsValid` function to a constant member function named `isValid` that takes a constant reference to a `Location` as a parameter instead of as two `int`s. The `isValid` function should return true if the specified location is inside the bounds of the `nodes` array and false otherwise.

    - **Hint:** Be careful that `Location` is passed by constant reference in both the prototype and the implementation. Also, made sure that the function is declared to be `const` in both places.

10. Convert the `worldIsDeath` and `worldIsVictory` functions to be constant member functions. Change the functions so that each function takes a constant reference to a `Location` as a parameter. Each of these functions should use an `assert` to make sure that the location is valid.

    - **Reminder:** You will need the `<cassert>` library.
    - **Hint:** Use the `isValid` function.

11. Convert the `worldCanGoNorth` function to a constant member function named `canGoNorth` that takes a constant reference to a `Location` as a parameter and uses an `assert` to make sure it is valid. Convert the `worldCanGoSouth`, `worldCanGoEast`, and `worldCanGoWest` functions in an analogous manner.

    - **Hint:** For the `canGoNorth` function, you may want to use a local variable to store the location that is one position to the north of the given location.

12. Add a constant member function named `getNorth` that takes a constant reference to a `Location` as a parameter. It should return the `Location` to the north of the specified location. It should use `assert`s to ensure that the location parameter is a valid location and that the player can go north from that location. Also add `getSouth`, `getEast`, and `getWest` functions.

    - **Hint:** For the `getNorth` function, you may want to use a local variable to store the location that is one position to the north of the given location.

13. Replace the `worldFindValue` function with a member function named `getStart`. The function logic should be similar, but it should always search for the `START_NODE` constant. Instead of modifying row and column values passed by reference, it should return the location it finds as a value of the `Location` type. If it does not find a node with that value, it should return a `Location` constant named `NO_SUCH_VALUE` with a value of `(-1, -1)`.

    - **Hint:** You may find it helpful to declare a local variable of the `Location` type.
    - **Note:** Declare the `NO_SUCH_VALUE` constant in `World.h`. Use the initializing constructor for the `Location` type to set it to have the value of `(-1, -1)`. (See the "Other Uses of Constructors" part of Section 7.1 of the online Notes.) Since the `getStart` function can return the special value of `NO_SUCH_VALUE`, this constant must be part of the interface. By declaring the constant in the interface file, you are allowing client code to use it when checking the returned value.

14. Convert the `worldPrintStartMessage` and `worldPrintEndMessage` functions to member functions.

15. Convert the `worldPrintDescription` function to a member function that takes a constant reference to a `Location` as a parameter. Use an `assert` to ensure that the location is valid.

16. Compile and link your `World` module with the `TestWorld3.cpp` program provided. You will also need your `Location` type and the `TestHelper` files. The test program should give you full marks.

    - **Reminder:** You will need create files named `ghostwood_grid.txt` and `ghostwood_text.txt` files and copy/paste their contents from the course website.
    - **Note:** You should have changed the `ROW_COUNT` and `COLUMN_COUNT` constants to `10` earlier. If you didn't, change them now.
    - **Hint:** `g++ Location.cpp World.cpp TestHelper.cpp TestWorld3.cpp`

17. Add `private` helper function named `isInvariantTrue` that takes no parameters and returns whether the class invariant is true The class invariant requires that:

    - `description_count <= MAX_DESCRIPTION_COUNT`
    - `descriptions[d] != ""`
      for all elements `0 <= d < description_count`
    - `nodes[r][c] < description_count`
      for all rows `0 <= r < ROW_COUNT`
      for all columns `0 <= c < COLUMN_COUNT`

    If any of these is false, the `isInvariantTrue` function should return `false`. If all the cases are true, the function should return `true`.

    - **Hint:** Organize your `isInvariantTrue` function with many `if(...) return false;` checks and a single `return true;` at the end.
    - **Hint:** Do <u>not</u> put any `asserts` inside the `isInvariantTrue` function.
    - **Hint:** If the isInvariantTrue function yields false during debugging, you should add print statements inside this function so that you can tell which of the three possible reasons is causing it to return false. Print the values of relevant variables in the function.

18. Check the class invariant at the end of every `public` member function that is not `const`. For the `World` class, this is just the constructor. At the end of that function, use an `assert` to make sure that `isInvariantTrue` returns `true`:

    ```
    assert(isInvariantTrue());
    ```

    The purpose of this is to ensure that the function does not leave the `World` in an invalid state. The `const` functions <u>cannot</u> change the internal state, so we do not need to check the class invariant at the end of these functions.

    - **Note:** Do not check the invariant in `private` functions. It will be checked in the functions that call them.

19. Check the class invariant at the start of every `public` member function except the constructor and `debugPrint`. Use an `assert` to make sure that `isInvariantTrue` returns `true`. In each case, the purpose of the check is to ensure that the `World` in a valid state when the function is called.

    - **Note:** We do not check the invariant in `debugPrint` so we can still print out the world fields if something goes wrong are we want to find out what.
    - **Reminder:** Do not check the invariant in `private` functions, such as `loadNodes`.

20. Add interface specification documentation for the `World` constructor using the style shown in the class notes. You will need a purpose, one parameter, and a side effect.

    - **Reminder:** Interface documentation is for written for other programmers who have your header file but not your source file. It should be placed in the interface file rather than the implementation file.
    - **Note:** Do <u>not</u> document the class invariant as a precondition.

21. Add documentation for the `getNorth` function in the same style. Make sure to include every parameter and every precondition.

Part C: Refactor `Item` into a Class [27% = 20% test program + 7% code]

Refactor the `Item` type to be an encapsulated class, and change it to use `Location`s instead of pairs of row and column values. The functions associated with the `Item` type will become member functions.

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

```
Item ();  // new in A3
Item (char id1,
      const Location& location,
      int points1,
      const string& world_description1,
      const string& inventory_description1);
void debugPrint () const;
bool isInitialized () const;  // new in A3
char getId () const;
bool isInInventory () const;
bool isAtLocation (const Location& location) const;
int getPlayerPoints () const;
void printDescription () const;
bool operator< (const Item& other) const;  // new in A3
void reset ();
void moveToInventory ();
void moveToLocation (const Location& location);
```

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

```
bool isInvariantTrue () const;  // new in A3
```

Perform the following steps:

1. Change the `Item` record (`struct`) to be an encapsulate `class`.
    - **Reminder:** In an encapsulated class, all member fields should be `private`.
2. Replace the `start_row`, `start_column`, `current_row`, and `current_column` fields with `start_location` and `current_location` fields of the `Location` type.
3. Add a default constructor to the `Item` class. It should use an initializer list to initialize the member variables. Initialize `id` to `ID_NOT_INITIALIZED`, initialize `is_in_inventory` to `false`, initialize `points` to `0`, and initialize both descriptions to `"[Item not initialized]"`. Initialize the starting and current locations by explicitly invoking the default constructor for the `Location` type. (For an example of how to use a constructor with an array, see the "Other Uses of Constructors" part of Section 7.1 of the online Notes.)
    - **Reminder:** An **initializer list** starts with a colon and appears after the first line of the function and before the function body. Here is an example:
```
MyClass :: MyClass ()  // default constructor
      : int_field(999),
        float_field(3.14159f),
        string_field("Initial value"),
        class_field()  // explictly use default constructor
{
      // ...
```
4. Convert the `itemInit` function to an initializing constructor for the `Item` class. Replace the row and column parameters with a single constant reference to a `Location`, and initialize both the starting location and current location to that value. Initialize `is_in_inventory` to `false`. Use an initializer list to initialize the member variables.
    - **Reminder:** Every constructor for a class must initialize every member variable of the class. In the `Item` class, each of the two constructors must initialize all seven variables.
5. Convert the `itemDebugPrint` function to a constant member function named `debugPrint` and update it to handle the `Location` fields. Each `Location` should be printed on a single line with the field name followed by a colon and a tab, followed by the field value.
    - **Hint:** Because you made an `operator<<` function for the `Location` type in Part A, it is easy to print values of the `Location` type: you can simply use the `<<` operator; for example:
```
cout << "my_location = " << my_location << endl;
```
    - **Warning:** The test program expects the `Location` to be printed in the format described in Part A. If you do not print in that format, it will probably mark your output as incorrect.
    - **Warning:** The test code for the `debugPrint` function has been revised since Assignment 2. Make sure your function implementation still works.

6. Add a constant member function named `isInitialized`. It should return `false` if the `id` field is `ID_NOT_INITIALIZED` and `true` otherwise.

   Also add a constant member function named `getId`. It should return the value of the `id` field.

7. Convert the `itemIsInInventory`, `itemGetPlayerPoints`, and `itemPrintDescription` functions to be constant member functions (with `item` removed from their names). Change the `itemReset` and `itemMoveToInventory` functions to be non-`const` member functions.

8. Convert the `itemIsAtLocation` function to a constant member function named `isAtLocation` that takes a constant reference to a `Location` as a parameter. Convert the `itemMoveToLocation` function in an analogous manner.

   - **Hint:** The `Location` type does not have the inequality operator (`!=`) defined. You can work around this by checking if it is not the case that two `Location`s are equal:

     ```
     if(!(location1 == location2))
     ```

9. Add a less-than operator (`operator<`) to the `Item` class. It should be a constant function that take a constant reference to another `Item` as a parameter. To compare two items, we will actually only compare their id fields. The operator should return whether the `id` for this current `Item` is strictly less than the `id` for the other parameter `Item`. You will use this function to sort the items in Assignment 4.

   - **Hint:** The less than operator is already defined for `char`s, so you can write `id < other.id`.

   - **Note:** You can make this operator a member function or a non-member function; both work. However, coding it as a member function is easier because you have access to the `private` member variables.

10. Compile and link your `World` module with the `TestItem3.cpp` program provided. You will also need your `Location` type and the `TestHelper` files. The test program should give you full marks.

    - **Hint:** `g++ Location.cpp Item.cpp TestHelper.cpp TestItem3.cpp`

11. Add `private` helper function named `isInvariantTrue` that takes no parameters and returns whether the class invariant is true. The class invariant requires that the world description is not an empty string and that the inventory description is not an empty string.

    - **Reminder:** Do not put any `asserts` inside the `isInvariantTrue` function.

12. Use an `assert` to ensure that the class invariant is true at the end of every `public` member function that is not constant. There are five such functions in the `Item` class.

    - **Reminder:** Constructors are functions. Think about whether you need to check the class invariant at the end of constructors.

    - **Reminder:** Don't check the invariant in `private` functions.

13. Use an `assert` to ensure that the class invariant is true at the start of every `public` member function except the constructors and `debugPrint`.

Part D: Update the `main` Function [23% = 8% stability + 15% test scripts]

Adapt your `main` function to use the refactored `World` and `Item` classes.

1. Replace the player row and column location variables with a single variable of the `Location` type.
2. Update the `World`-related functions to use dot notation. They should also use `Location`s instead of rows and columns. Use the `World` constructor instead of declaring the `World` and initializing it later. Also use the `getNorth`, etc. functions to determine the location to go to, instead of incrementing or decrementing the row and column values.
    - **Example:** `worldPrintStartMessage(my_world)` should become `my_world.printStartMessage()`
    - **Hint:** When you call a function such as `getNorth` that returns a value you should save the value it returns in a variable. Example statements:

```
my_world.getNorth( );      // useless because returned value is ignored

my_loc = world.getNorth( ); // saves returned value in a variable
```

3. Update the `Item`-related functions to use dot notation and `Location` variables.
4. Do not access the fields of the `Location` type anywhere in the `main` function.
5. Load a new game world named `ghostwood`. The `ghostwood_grid.txt` and `ghostwood_text.txt` files are on the course website.
    - **Reminder:** You already changed the. `ROW_COUNT` and `COLUMN_COUNT` constants to both be `10` in Part B.
6. At this point, your game should run again.
7. Replace the current items with a new set of nine different items and update the `ITEM_COUNT` constant.
    - A scarab beetle with id `'s'` at location row `0` and column `3`, worth `-5` points, with world description `"There is a black scarab beetle (s) here."` and inventory description `"A black scarab beetle (s) is crawling up your arm."`.
    - A candlestick with id `'c'` at location row `1` and column `1`, worth `9` points, with world description `"There is a silver candlestick (c) here."` and inventory description `"You are carrying a silver candlestick (c)."`.
    - A key with id `'k'` at location row `2` and column `0`, worth `3` points, with world description `"There is an old iron key (k) here."` and inventory description `"You have an old iron key (k) in your pocket."`.

- A tarantula with id `'t'` at location row `2` and column `9`, worth `-8` points, with world description `"There is a tarantula (t) here."` and inventory description `"There is a tarantula (t) hanging on your shirt."`.
- A book with id `'b'` at location row `3` and column `4`, worth `4` points, with world description `"There is a book (b) here with an eye drawn on the cover."` and inventory description `"You have a book (b) under your arm with an eye drawn on the cover."`.
- A moth with id `'m'` at location row `5` and column `5`, worth `-2` points, with world description `"There is a giant moth (m) sleeping here."` and inventory description `"A giant moth (m) is perched on your shoulder."`.
- An amulet with id `'p'` at location row `7` and column `9`, worth `7` points, with world description `"There is a golden pendant (p) here."` and inventory description `"You are wearing a golden pendant (p)."`.
- A dagger with id `'d'` at location row `8` and column `0`, worth `1` points, with world description `"There is an rune-carved dagger (d) here."` and inventory description `"You have an rune-carved dagger (d) stuck in your belt."`.
- A ring with id `'r'` at location row `9` and column `6`, worth `10` points, with world description `"There is a diamond ring (r) here."` and inventory description `"You are wearing a diamond ring (r)."`.

8. Test your program with the five test cases provided: `testcase3A.txt`, `testcase3B.txt`, `testcase3C.txt`, `testcase3D.txt`, and `testcase3E.txt`.
    - **Remember:** Each test case is a series of commands you should enter to your program in order. The easiest way to do this is to copy/paste everything from the file into the console.

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. **`Item.h`**
  2. **`Item.cpp`**
  3. **`Location.h`**
  4. **`Location.cpp`**
  5. **`main.cpp`**
  6. **`World.h`**
  7. **`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