Assignment 2

Overview

A simple exploration game such as the one in Assignment 1 may be fun for a short time, but the lack of challenges limits its appeal. The most common solution to this is to include obstacles in the world that can only be overcome in a specific way. Failure to overcome an obstacle can have varying penalties, ranging from inability to continue (the player can't get past the locked gates of the castle) to automatic loss of the game (the player is killed by the castle guards). One of the simplest types of obstacle is one that the player can only pass after acquiring a specific item (such as the key to the castle) earlier in the game. In this assignment, you will make an encapsulated C++ record representing an item for a game similar to the one in Assignment 1. There will be no actual obstacles in this assignment; those are in Assignment 6.

Each item will be identified by a single-character id, such as `'a'` for apple and `'b'` for boat. Each item can be in the player's inventory or at any location in the world. Whenever the player is at a location, the game will print a list of any items there. The player will have new commands to take an item (`'t'`), leave an item (`'l'`), and display the inventory (`'i'`).

The purpose of this assignment is to ensure that you understand how to use C++ records, including passing records as parameters, and how a record can be used to implement an abstract data type. For Part A, you will develop an `Item` module to represent an item in the game. For Part B, you will incorporate your `Item` module into your game program.

Copy the code and data files of your Assignment 1. Do not just modify Assignment 1.

(If you are using Visual Studio, you must start by creating a new project for Assignment 2. Do NOT copy the whole folder including the `.sln` file or massive confusion will result!)

Requirements

Part A: The `Item` Type [70% = 55% test program + 5% code + 10% documentation]

Create a record named `Item` to represent an item in the world. Put the `Item` type and its associated constants and functions in its own module with its own interface and implementation files.

HINT: `Item.cpp` and `Item.h` should now exist

The `Item` type should be treated as if it is encapsulated, i.e., do not access any of its variables outside the `Item.cpp` file.

By the end of Part A, your `Item` type will have associated functions with the following prototypes:

```
•  void itemInit (Item& item,
•                 char id1,
•                 int row1, int column1,
•                 int points1,
•                 const string& world_description1,
•                 const string& inventory_description1);
```

- **`void itemDebugPrint (const Item& item);`**
- **`char itemGetId (const Item& item);`**
- **`bool itemIsInInventory (const Item& item);`**
- **`bool itemIsAtLocation (const Item& item,`**
- **`                       int row, int column);`**
- **`int itemGetPlayerPoints (const Item& item);`**
- **`void itemPrintDescription (const Item& item);`**
- **`void itemReset (Item& item);`**
- **`void itemMoveToInventory (Item& item);`**
- **`void itemMoveToLocation (Item& item,`**
- **`                         int row, int column);`**

Perform the following steps:

1. Define `Item` as a record (`struct`) with the following fields:

    - A `char` representing the item id (e.g. `'a'` for apple)
    - A pair of `int`s representing the starting row and column
    - A pair of `int`s representing the current row and column
    - A `bool` representing whether the item is in the player's inventory
    - An `int` representing how many points the item is worth (this value can be negative)
    - A pair of `strings`s storing the item's description when it is on the ground and when it is in the inventory

2. Add a `char` constant named `ID_NOT_INITIALIZED` with a value of `'\0'`.

3. Add stand-in implementations for the `Item` functions. Functions that do not return values should have empty function bodies. Functions that return a `bool` should return `false`, functions that return a `char` should return `'\0'`, and functions that return an `int` should return `0`.

4. Compile and link your `World` module with the `TestItem2.cpp` program provided. You will also need the `TestHelper.h` and `TestHelper.cpp` files. Run the resulting program. It should give you a mark of `27 / 55`.

    - **Hint:** `g++ World.cpp Item.cpp TestHelper.cpp TestItem2.cpp`

5. Add code to the `itemInit` function, which initializes an `Item`. As parameters, it should take a non-constant **<u>reference</u>** to an `Item`, a `char` representing the id, two `int`s representing a location (row and column), an `int` representing the points the `Item` is worth, and two `string`s representing the descriptions used when the `Item` is in the world and when it is in the player's inventory. Initialize the item fields to these values. Both the starting and current locations should be set to the row and column provided, and the item should be set to not be in the inventory. Use `assert`s to ensure that the id parameter is not `ID_NOT_INITIALIZED` and that neither description parameter is an empty string.

6. Add code to the `itemDebugPrint` function, which takes a constant reference to an `Item` as a parameter and then prints all fields of the item. Each field should be printed on its own line with the format `field_name: value`. Print single quotes (`'`s) around the id and double quotes (`"`s) around the descriptions. Print a colon (`':'`) and a tab (`'\t'`) between the field name and its value. Do not print any other tabs. Possible correct output would include:

```
id:     'a'
start_row:      4
start_column:   2
current_row:    1
current_column: 7
is_in_inventory:0
points:         5
world_description:      "There is an apple (a) here."
inventory_description:  "You are carrying an apple (a)."
```

   - **Hint:** When you are debugging your program, you can call the `itemDebugPrint` function whenever you want to know whether any of the functions in the module is behaving correctly. For example, call `itemDebugPrint` at the beginning of a function and then call it again at the end or return-point of the function and see if a proper change has been made.

7. Add code to the `itemGetId` function, which takes a constant reference to an `Item` as a parameter and returns the item's id value.

8. Add code to the `itemIsInInventory` function, which takes a constant reference to an `Item` as a parameter and returns a Boolean value that tells whether the item is in the player's inventory.

   - **Hint:** You can check the field in the `Item` that says whether or not the item is in the player's inventory.

9. Add code to the `itemIsAtLocation` function, which takes a constant reference to an `Item` and a row and column representing a location as parameters. The function should return whether the item is at that location. An item in the player's inventory is not at any location, no matter what its row and column values are.

   - **Hint:** Start by checking if the item is in the player inventory and, if so, return `false`. If not, check the location.

10. Add code to the `itemGetPlayerPoints` function, which takes a constant reference to an `Item` as a parameter and returns how many points the player current has from the item. If the item is in the player's inventory, the function should return the value of `points` field. If the item is not in the player's inventory, the function should return `0`.

11. Add code to the `itemPrintDescription` function, which takes a constant reference to an `Item` as a parameter. If the item is in the player's inventory, the function should print the inventory description; otherwise it should print the world description.

12. Add code to the `itemReset` function, which takes a non-constant reference to an `Item` as a parameter. It should move the item to its starting location and set it to not be in the player's inventory.

13. Add code to the `itemMoveToInventory` function, which takes a non-constant reference to an `Item` as a parameter and moves the item to the player's inventory.

14. Add code to the `itemMoveToLocation` function, which takes a non-constant reference to an `Item` and a row and column representing a location as parameters. It should move the item to the specified location and set it to not be in the player's inventory.

15. Test your `Item` module using the `TestItem2.cpp` program again. At this point, it should compile, run without crashing, and award you full marks.

16. Add three preconditions to your `itemInit` function. Each should be enforced with an `assert` at the beginning of the function body. The first precondition is that the item id parameter is not `ID_NOT_INITIALIZED`. The second and third preconditions are that the two description parameters are not the empty string (`""`).

    - **Hint:** You will need the `<cassert>` library.

    - **Reminder:** You can compare strings with the == and != operators.

17. Add documentation for the `itemGetPlayerPoints` function using the style shown in the class notes for the Program Organization lecture. You will need a purpose, one parameter, and a return value.

    - **Reminder:** Interface documentation is written for other programmers who have your header file but not your source file. They want to know what your function does and how to use it. They can see the function prototype but not the function implementation. Thus, they can see the name and type of a parameter but not what it is used for.

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

    - **Hint:** A good way to document a precondition is to copy the check from inside the `assert` statement into the documentation.

Part B: Add Items to the Game [30% = 10% stability + 20% test scripts]

In Part B, you will improve your game (in `main.cpp`) to handle items.

About compiling:

If you are working in Replit, you can use the following to compile in Part B:

```
g++ main.cpp Item.cpp World.cpp -o game
```

If you are working in Visual Studio, you should exclude `TestHelper.cpp` and `TestItem2.cpp` from your project and add `main.cpp`.

1. Add a constant named `ITEM_COUNT` to your `main` function with a value of `5`. Add an array of `ITEM_COUNT` `Item`s and initialize them as follows:

    - A girl named Alice with id `'a'` at location row `0` and column `6`, worth `1` point, with world description `"You see Alice (a) here, trying to read a very small compass."` and inventory description `"Somewhere nearby, you hear Alice (a) jabbering about directions."`.

    - A boy named Charlie with id `'c'` at location row `0` and column `6`, worth `1` point, with world description `"You see Charlie (c) lying half-buried in the drifting snow here."` and inventory description `"Behind you, Charlie (c) is dragging himself through the snow."`.

    - A girl named Emma with id `'e'` at location row `3` and column `5`, worth `1` point, with world description `"Young Emma (e) is crouched down here, out of the wind."` and inventory description `"Young Emma (e) is trying to use you to block the wind."`.

    - A boy named David with id `'d'` at location row `4` and column `1`, worth `1` point, with world description `"David (d) is using a stick to write in the snow here."` and inventory description `"David (d) is dashing this way and that, despite the weather."`.

    - Twin boys with id `'b'` at location row `5` and column `8`, worth `2` points, with world description `"The twins, Benny and Bobby (b), are huddled together here."` and inventory description `"The twins, Benny and Bobby (b), are huddled behind you."`.

2. The code from Assignment 1 prints the description of the node (this may be done in several places in the code.) In every case where you print the node description, also print the descriptions of any items at the player location. Recall that the player location is given by a combination of a row and a column.

    - **Hint:** Write a helper function that uses a FOR loop to find every item at a location and print their descriptions. This function can be placed in `main.cpp`. Call this function immediately after you call the function that prints the node description.

3. Add a command `'t'` to take an item. When the player enters the take command, the game should print `"Take what? "` without a newline and read in another line of player input. If the first character of that line is the id of an item at the player location, that item should be moved to the inventory. Otherwise, the game should print `"Invalid item"`.

    - **Hint:** Remember that if the player enters a blank line (empty string), it does not have a first character. In this case, your program should print `"Invalid item"` and not crash.

- **Hint:** You can use a FOR loop to check every item to see if it is the item that the player wants to take. If so, check whether it is at the player location. You may want to use a Boolean variable to keep track of whether the item was available for taking.

- **Hint:** You can declare variables inside a `switch` statement if you put curly braces `{}` around the part of the code where they are used. Put the opening brace `'{'` after the `case` statement and the closing brace `'}'` before the `break` statement.

4. Add a command `'l'` to leave an item, similar to the take command. The symbol `'l'` is a lowercase L. The game should print `"Leave what? "` before reading a line of player input. If the first character of the line is the id of an item in the inventory, that item should be moved to the player location. Otherwise, the game should print `"Invalid item"`.

5. Add a command `'i'` to display the inventory. The game should print the description for each item in the player's inventory.

6. When the game ends, print the player score. The player score is the sum of the points that the player receives for each item. A suitable message would be `"In this game you scored ??? points."`, where `???` is the player's score.

7. Test your program with the five test cases provided: `testcase2A.txt`, `testcase2B.txt`, `testcase2C.txt`, `testcase2D.txt`, and `testcase2E.txt`.

   - **Hint:** `g++ World.cpp Item.cpp main.cpp`

   - **Remember:** Each test case is a series of commands you should enter into 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. **`main.cpp`**
    4. `World.h`
    5. `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. 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