Project 2: Autotesting Gold Chase

The purpose of the second project is to develop a driver program that will automate evaluation of the Gold Chase game. The driver will automatically start two game processes (that is, two players), provide input to the game processes, and evaluate the game output for correctness. To facilitate the automation, the implementation of the `Map` class will be replaced with code that writes/reads game data that the driver program evaluates.

As a first step to understanding where the Gold Chase game could exchange information with the driver program, we need to identify the points where the game currently interacts with the `Map` class:

1. Open `mymap.txt` (or some other filename), read in the number of fools/gold and the lines of the map.
2. Call `Map` constructor, e.g., `renderer=new Map(mapMemory->board,rows,cols);`
3. Call `Map::getKey()`
    1. interpret input, update shared memory if necessary
    2. call `Map::drawMap()` -- if appropriate
    3. call `Map::postNotice()` -- if appropriate
4. Repeat, until user wins or quits

In addition to interacting with a reimplemented `Map` class, the driver program should control the map that Gold Chase reads in--providing the game with a map which eases the driver's task of testing the game.

To simplify communications between the game's reimplemented `Map` class and the driver, communications should be restricted to two functions: `Map's` constructor and `Map`'s `getKey` member function. The constructor is ideal for providing information back to the driver--a kind of initial handshaking to confirm that everything is working properly. The `getKey` member function should pass information regarding the current state of the game. The other two member functions (`drawMap` and `postNotice`) can simply maintain counters of number of times called. These counters could then be passed to the driver as part of the state information that `getKey` provides.

Here's a good scheme for the `Map` member functions:

- `Map::Map` - write the rows, columns, and map to a pipe
- `Map::drawMap` - add one to a draw map variable
- `Map::postNotice` - add one to a post notice variable
- `Map::getKey` - write the "draw map" variable, "post notice" variable, and map to a pipe. Read a character from a 2nd pipe. Return the character from the function.

Before the driver can start up a Gold Chase game process, it must first establish a a communications pipeline.

1. The driver must be able to read from, and write to, a game process. Since a `pipe()` command creates a one-way pipe, create two pipes for a game process.
2. Fork the process.
3. Don't forget to have child and parent close all unnecessary file descriptors.

Once the pipelines have been prepared between the parent and child, the game can be exec'd in the child. The parent must now have the complementary behavior to that of the child described in the numbered list above:

4. Open `mymap.txt` (or some other filename), write out the number of fools/gold and the lines of the map. Close.
5. Read the rows, columns, and map from the game process (sent from `Map::Map`). Confirm expected values.
6. Read the "draw map" variable, "post notice" variable, and map from the game process (sent from `Map::getKey`). Confirm values are as expected (if necessary). Decide on what the next move should be and write a character to the game process.
7. Repeat #6 until test(s) are completed. If necessary, write a 'Q' to the game process to end it.
8. Note that the above description must be expanded to handle two game processes.

Working towards a completed project

Before and during coding of any large project, seasoned programmers ensure they are competent in the tools (system calls, programming structures, algorithmic flows, etc.) they are using. To ensure this competency, it is typical to write many small programs which serve as experiments with the various constructs in order to gain better understanding.

Below is one possible set of test programs which could be written to build competency in many of the tools required for this project:

1. Write a program which forks then execs your game. Have the parent call `waitpid()` to wait for the game to finish, then print a message to the screen.

2. Write a program which creates a named pipe (system call: `mkfifo`), then forks a child process. The parent should open and write to the named pipe, the child should open and read from the named pipe.

3. Write a program which creates an unnamed pipe (system call: `pipe`), then forks a child process. The child should close stdin and duplicate the input portion of the pipe (system call: `dup`), the parent should write to the output portion of the pipe. The child should then read from stdin (the now duplicated pipe) and print out what is sent. Don't forget to have the parent and child close unused file descriptors.

4. Enhance the above with a second pipe which replaces the child's stdout. Try sending messages from parent to child, then the other direction--from child to parent.

5. Enhance the above. After the child has established a 2-way pipeline with the parent, have the child exec the `sort` command. Have the parent write some words to the child (each ending with a newline), then read them back from the child, confirming the words are now in sorted order.