Holes Missed Without Static Debugging and Linearity

An issue facing every software developer is how to develop stable, extensible code in a short period of time that will be forwards compatible. By looking at holes in C, ways they might typically get filled, and advanced techniques, it will be shown how compile time verification and debugging are both good for producing extensible, forwards compatible code and can save development time.

Why is Tetris a good test for the merits of program verification?

Since Tetris is a game, it has several distinct elements that come together into the whole, but since it is not as complex as modern games, it still allows observers to peer into its mechanics with a decent amount of clarity. The main elements in focus will be the game grid, which is where small cells are filled and cleared when the row is full, and the tetromino, the shapes that fall in the game grid and that the player actively controls. Their definitions are fairly simple:

grid : in this example the grid is a simple array of integers. In grid.h, anytime it is used as an input there is an int* for it. Though a linear array, it is used to represent the 2 dimensional grid via the MAXCOL and MAXROW defines.

tetromino: a fairly simple struct -

```
typedef struct{
   int active;
   int shape;
   int orientation;
   int pivot_x;
   int pivot_y;
   int rot1_x;
   int rot1_y;
   int rot2_x;
   int rot2_y;
   int rot3_x;
   int rot3_y;
} tetromino
```

Where active tells us if the block is falling or stored, the shape tells us if it is long or s-shaped, the orientation tells us the number of turns, and the remaining integers tell us where each of the sub blocks in it go, where all of them rotate around the pivot.

Where are the weaknesses?

Immediately weaknesses are clear. In the tetromino, there is nothing programmatic enforcing the position of the sub-blocks, so it would be feasible in a later implementation to have an x or y declared that goes beyond the row or column limit imposed on the grid. In the lenient case, this will lead to a block being misprinted on the screen such that it might float. In the worst case, a crash during runtime will occur because of potentially bad memory access.

This can be clearly illustrated in the grid_coords_2_offset() funciton in grid.h:

```
int grid_coords_2_offset(int row, int col)
{
    return ((row * MAXCOL) + col);
}
```

It is completely possible to pass row and column indices that are far outside of the limits implied by MAXCOL and MAXROW. Since this is frequently used to calculate the offset into the grid, the programmer risks crashes or bad date through this.

The fix that comes quickly to mind for this is to use control structures influence the return, which would lead to something like this, potentially:

```
int grid_coords_2_offset(int row, int col)
{
      if(row >= MAXROW)
            return -1;
      if(col >= MAXCOL)
            return -1;
      return ((row * MAXCOL) + col);
}
```

Like this, the function now reports errors to the caller; however, this leaves another fix to be done. The caller code must also now be modified to handle the -1 returns. If that -1 is returned and passed to the grid array, there is again high risk. The expected result would be an invalid access, but even worse would be if the pointer for the array had been manipulated such that the negative subscript becomes valid, at which point it may return garbage and be even more difficult to trace.

Another resolution is to exit the process, such as:

```
int grid_coords_2_offset(int row, int col)
{
      if(row >= MAXROW)
            exit(0);
      if(col >= MAXCOL)
            exit(0);
      return ((row * MAXCOL) + col);
}
```

In practice, this could be combined with writing to a log file or terminal, but it again forces debugging on run time. Instead of actively searching out errors before they occur, they need to be found, potentially by luck, in order to be routed.

How can static debugging and types address this?

Static debugging and a type system can be used to push finding these bugs up to compile time rather than at run time. From the example above, and other functions in grid.h, we can constrain indexing inputs to MAXCOL or MAXROW and the array to be strictly a grid of MAXROW*MAXCOL elements, other inputs that disobey this typology are disallowed. For example, in the grid_coords_2_offset, we could constrain row from zero to less than MAXROW and col from zero to MAXCOL. This then ensures the offset will be in the bounds of the grid array. If such a constraint could be implemented in this C, at compile time for the whole project, each call to this function would raise an error that would need to be fixed. The types of arguments being passed to it would be forced to obey the new type parameters, which ensure desired behavior. Effectively, the debugging end of this has been moved to compile time, so that when it executes a user should not have to deal with poor execution. Any further changes to this would require the same systematic changes, but the advantage is that they can be detected before they occur and they can be detected easily and patched quickly and safely. This is important to note since it is typically in patching code that most bugs get introduced and missed.

What other techniques can strengthen issues seen here?

Looking to the grid, if we view each element as a linear resource and base our usage on the consumption and production of each element, we can further ensure good behavior in Tetris. For example, as the tetromino falls, it must consume the positions it is taking, and produce the ones it is leaving. When it can no longer move, the focus shifts to the next one. The sheer act of falling and checking if it can keep falling means it is using a resource, and when the focus shifts to the next falling piece, it will not be able to consume what the last one had taken.

This system of consumption is safer than the checking and locking mechanisms found in tetromino_lock() and tetromino_into_grid(). First, since there are a limited number of places to consume, it can be ensured that only places in the grid can be moved to and consumed. The tetromino_into_grid() function cannot mimic this easily. Currently, it only sets the desired positions high. If one was already high, it won't be easily noticed. Again, even if control structures are used here, the programmer must rely on runtime behavior. This is less likely to happen with linear resourcing since the tetromino would not have been able to consume a preconsumed position and then be able to accidentally set it.

Another potential benefit to linearity can be seen quickly for the tetrominos. If treated as a linear resource, the programmer typically will no longer have to remember to deallocate memory for it, and the programmer would not have to worry about the pointer being copied illicitly since it could only be moved.

Why these are of use

By using static debugging, it becomes quickly clear that we can guarantee the functionality of software before running it. It also allows for narrowing of definitions for better behavior or more controlled behavior. This means that while the building of software may take longer to get to an initial compile, once it is working the programmer can be sure that further reasonable restrictions imposed will allow for safe patching and guaranteed performance.

While trivial in an application like Tetris, it is clear that the benefits of static debugging can be seen. In a more dire setting, such as software running at hospitals, it can be life changing. A fix doesn't mean a serious risk of the wrong medication being delivered or a server crashing from an invalid memory access. This guarantee for patching not only would serve to ensure functionality and correctness, but it would also serve peace of mind in such a setting when changes are needed in software.