

Assessment Report CUP455

15618218_Graham_Riley_DAC416_AE1_Report

Input Output

So the goal is to create a turn-based game using the console for input and output. The user (player) has the whole keyboard to use but to keep complexity to a minimum I decided to go with numbers, ranging from 1 to 6, most keyboards have a very similar setup to each other, being a straight line of numbers so it made sense. With this control scheme, you could also do interesting setups, such as using a random number generator for input, since it's all integers, which is much easier to set up compared to random characters.

Output is also crucial in conveying information, it's the only way to give feedback on actions which is why I spent a lot of time making the output as straightforward as possible and refreshing the screen to avoid clutter. Text is the obvious option for outputting information but not the only one, however in the console only text can be outputted but this CAN be used to create images, such as ASCII art. For the simplicity of this project, I just used regular text.

Below is an image of my game running in a terminal/command prompt. There were quite a few design choices I went with, so I'll list them and the reasons for them.

```
G:\not linux lol\Please no\6- x + v
Fight!

Player HP:98      Enemy HP:100
Player Energy:60  Enemy Energy:10

Press 1 for a regular attack (has 80% hit chance)
Press 2 for a special attack (has 50% hit chance)
Press 3 to recharge energy (recharges energy 4 times faster for this turn, but gives enemy 10% higher hit chance)
Press 4 to dodge (decreases enemy hit chance by 30%, but halves energy recharge rate)
Press 5 to heal (heal half your energy up to max health, and gives you chance to use another action)
Press 6 to exit the game

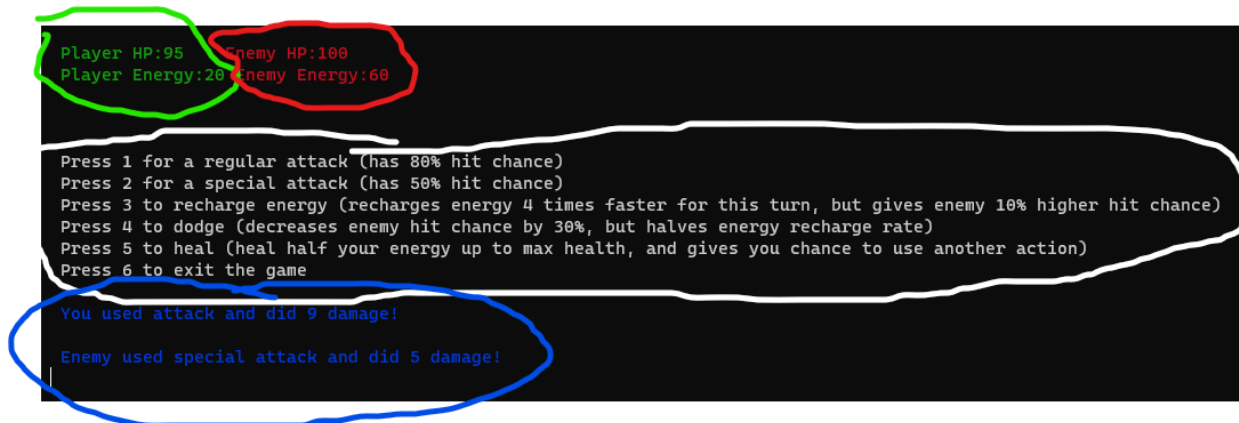
You used attack and did 7 damage!

Enemy used special attack and did 2 damage!
```

Colour: I chose to increase the complexity of my code and use a separate library in order to give

clearer feedback to the player, using colour to convey certain elements, with the actions being in white, direct feedback in blue, friendly (green) HP and energy and enemy (red) HP and energy. I believe this helps the user understand and take in information at a glance. Plus it looks pretty cool.

Layout: I chose to lay out my text in a particular way for a reason, there was a certain amount of care that went into making it visually appealing for a game that runs in a console. For example, all text has a margin of one space to the right, so that it doesn't hug the window, it keeps the game inside the window instead of on the border of it. Alongside this, I made sure that everything is in its own well-defined section, with whitespace being used to separate sections. I believe this makes it easier to read and take in the information without being overwhelmed.



The screenshot shows a console window with a black background. At the top, player and enemy stats are displayed: 'Player HP:95' and 'Player Energy:20' in green, and 'Enemy HP:100' and 'Enemy Energy:60' in red. These are circled with green and red lines respectively. Below this is a list of actions in white text: 'Press 1 for a regular attack (has 80% hit chance)', 'Press 2 for a special attack (has 50% hit chance)', 'Press 3 to recharge energy (recharges energy 4 times faster for this turn, but gives enemy 10% higher hit chance)', 'Press 4 to dodge (decreases enemy hit chance by 30%, but halves energy recharge rate)', 'Press 5 to heal (heal half your energy up to max health, and gives you chance to use another action)', and 'Press 6 to exit the game'. At the bottom, two lines of blue text provide feedback: 'You used attack and did 9 damage!' and 'Enemy used special attack and did 5 damage!'. These lines are circled with a blue line.

```
Player HP:95      Enemy HP:100
Player Energy:20  Enemy Energy:60

Press 1 for a regular attack (has 80% hit chance)
Press 2 for a special attack (has 50% hit chance)
Press 3 to recharge energy (recharges energy 4 times faster for this turn, but gives enemy 10% higher hit chance)
Press 4 to dodge (decreases enemy hit chance by 30%, but halves energy recharge rate)
Press 5 to heal (heal half your energy up to max health, and gives you chance to use another action)
Press 6 to exit the game

You used attack and did 9 damage!
Enemy used special attack and did 5 damage!
```

Refreshing the screen: Normally when you output text to the console, it appends it, it adds on from the bottom after the last bit of text, and this isn't ideal if all you're trying to do is update values, such as changing the HP of the enemy. So it would make sense to edit the text, right? Unfortunately, there's no good solution to edit already displayed text without a lot of added complexity. So my solution is to clear the screen every turn and then output text. From the user's perspective it looks like the text has been *changed*, not completely *replaced*. If this solution wasn't in place, the screen would quickly get cluttered and the user would have greater difficulty determining their place in the console.

Testing

No program is perfect, mine is no exception, there are always going to be bugs or ways to reach code that shouldn't be accessible but for smaller programs like mine, you can do a pretty good job of making it damn near bug-free. Throughout development, I tested my game extensively, after every change and just routinely playtesting to make sure it runs well and plays well. I found many bugs. There was a bug that locked the user out of input and repeatedly outputted the exact same piece of text, and it took me ages to figure out that it was because I had a loop I

wasn't breaking out of after I took input from the user. Expected behaviour was to take input, use the input in the decision for user action, perform the action, clear the screen, output text and wait for user input. Actual behaviour was very much not that. I debugged the code by first looking for exceptions, which gave me no new information, so then I used breakpoints to identify the problem and used an iterative process to locate the exact cause from that section of code (I changed random values until I understood where I went wrong). The solution was to change the code that controlled the loop. I used this process throughout the entire project and it ensured that bugs that I introduced when adding new features or tweaking values then didn't add up and were just fixed very quickly.

Design

When I was given the project I immediately had a few ideas of how the code was going to be structured and what logic I needed but I found myself getting lost in specifics and not being able to focus on the big picture and while it led to good specific features such as coloured text, it meant I wasn't prioritizing the important things, like the game loop which was a massive problem. I had started working on a very rough proof of concept before I had started making any sort of design plans and then got too involved in it, so refocused on getting the core ideas down in a design document, using both flowcharts and pseudocode. I struggled with the design process, I found it very hard to make these ideas abstract and not think in terms of code and instead think in terms of problems and solutions. The image below showcases the very codelike nature of what is *supposed* to be a design-based document. I have work to do in this aspect, design documents are supposed to be abstract for a few reasons and the lack of abstraction is a major problem.

However, there are clearly laid out solutions to problems and the logic is solid, it's just too specific and too rigid, which I will work on.

```

start

initialize all functions and variables/constants

main function
  initialize and set "finished menu" variable to false
  while "finished menu" is false {
    initialize and clear "menu choice" value
    display main menu screen and choices
    wait for input and change "menu choice" value to inputted value

    if "menu choice" is out of defined range, display wrong key message and break
    if input is play button, start function to start playing (eg. play function)
    if input is settings button, start function to start settings screen (eg. settings function)
    if input is exit button, set "finished menu" variable to true and exit program
  }

play function
  initialize all variables/constants for player/enemy (ie. healthpoints, energy)
  initialize empty variables for input
  initialize "game over" variable and set to false

  while "game over" is false {
    initialize temporary values for player/enemy (ie. hit chance, hit damage range, energy recharge rate and energy recharge multiplier)

    if player healthpoints is below 1, set player HP variable to 0
    if enemy healthpoints is below 1, set enemy HP variable to 0

    display fight menu on screen along with menu choices and things like player/enemy variables (ie. healthpoints and energy)

    if player healthpoints is below 1, display "you lose" message on screen, set "game over" variable to true (ending the while loop) and return
    if enemy healthpoints is below 1, display "you win" message on screen, set "game over" variable to true (ending the while loop) and return

    if "game over" variable is false, wait for user input

    //player action logic
    if input is attack, calculate random hit chance and subtract from hit chance variable, also calculate hit damage range and put that into varia
    if input is special attack

```

Now, to the actual design portion. There were quite a few problems to solve before starting on the main functions, such as the game loop, how I was storing information and obviously things I've mentioned above, like input and output.

First I needed to figure out how I was managing looping the game since it needs to "loop" every turn.

The player needs to be presented with options to choose from

The player picks an option

The enemy picks an option

All the math happens

Game checks such as if someone's health is too low

The player is then presented with options and feedback based on their action

I really struggled to get that in the right order when I was first working on it, the player would pick their option, I would do the math and random chance and THEN the enemy would pick an option, which was convoluted in practice and required a major redesign.

I am quite happy with how I store my data throughout my program, the things that need to be constant are public constants, the variables have a very small scope, declared, reset and used exclusively in the loop.

```

d play() {
    // "permanant" variables, doesn't get reset each round
    int startHP = 100;
    int startENG = 50;

    int charHP = startHP;
    int enemHP = startHP;
    int charENG = startENG;
    int enemENG = startENG;

    int specialAttackChance = 8;
    int attackChance = 5;

    char anything;

    bool gameOver = false;
    bool firstTurn = true;

    while (!gameOver) {
        // "temporary" variables, does get reset each round
        int charHitChance = 0;
        int enemHitChance = 0;
        int charHitDamageRange = 0;
        int enemHitDamageRange = 0;
        int charHitChanceModifer = 0;
        int enemHitChanceModifer = 0;
        int charEngRechargeRate = 10;
        int enemEngRechargeRate = 10;
        int EngRechargeMult = 4;
        int playMenuChoice;
        bool exitConfirm = false;
        std::string exitConfirmVar;
        std::string charResult;
        std::string enemResult;
        std::string charDamage;
        std::string enemDamage;
    }
}

```

I also make use of checks to make sure unexpected behaviour doesn't happen, this includes

```

// user input validation
std::cin.clear();
std::cin.ignore(std::numeric_limits<char>::max(), '\n');
system("CLS");

```

user validation and

```

// clamp energy to 100
if (charENG > 100) { charENG = 100; }
if (enemENG > 100) { enemENG = 100; }

firstTurn = false;

// round health to 0 if below 1
if (charHP < 1) { charHP = 0; };
if (enemHP < 1) { enemHP = 0; };

```

clamping of values.

The program decides who's won and who's lost based on health, if health is 0 or below (negative numbers are impossible because of the above code, but for completeness...).

```

// score check
if (charHP < 1) {
    std::cout << FYEL(" \nYou lose!\nPress 1 to go to menu!\n");
    std::cin >> anything;
    gameOver = true;
    return;
}

if (enemHP < 1) {
    std::cout << FYEL(" \nYou win!\nPress 1 to go to menu!\n");
    std::cin >> anything;
    gameOver = true;
    return;
}

```

And the enemy's action is based on random chance, using a random number generator and bias for certain actions, for example, the heal action is favoured just to make it a little harder.

```

switch (rand() % 6) {

    // attack
case 0:
case 1:
    enemHitChance = rand() % 10 + 1;
    enemHitDamageRange = rand() % 10 + 1;
    enemResult = "attack";
    enemSuccess = true;
    enemHealChosen = true;
    break;

    // special attack
case 2:
    if (enemENG >= 50) {
        enemENG -= 50;
        enemHitChance = rand() % 10 + 1;
        enemHitDamageRange = rand() % (20 + 1 - 5);
        enemResult = "special attack";
        enemSuccess = true;
        enemHealChosen = true;
        break;
    }
    else {
        enemSuccess = false;
        break;
    }
    break;
}

```

In conclusion,

I had difficulty with certain aspects of the design process but eventually, through iterative development, I refined both my design and implementation. I went through quite a few refactors, going from everything being in one function to everything split into specific functions for specific things, which aided readability and functionality. I found, and fixed, multiple bugs and was left with a nearly bug-free turn-based console game. Thank you for reading :)