Task 1

To simplify and keep the code tidy, I wrote the ASCII-art as a multi-line string, rather than `cout`-ing many individual lines. Since backslash is treated as the indicator of an escape character, I had to replace each backslash with a double backslash in order to produce a single backslash in the output. I found this task easy, as I'm already very familiar with C++ syntax. The ASCII art text was generated using https://patorjk.com/software/taag

Task 2

This task is mostly just making use of `cout` and `cin` as well as the `<<` and `>>` stream operators to assemble multiple strings into a single console line. I made sure to use `endl` rather than just `"\n"` because it reads more clearly and it ensures consistency across platforms (`CRLF` vs `LF` vs `CR`). I used the standard library function `int stoi(string)` to convert the age text input from the user into an integer. I again found this task easy since I'm already familiar with C++. I later came back and added better input handling, using `getline` instead of just `cin >>`, and wrapping `stoi` conversions in `try {} catch {}` blocks.

Task 3

This task is similar to the previous task in that it primarily involves getting input from the user via `cin`, though this time I used `float stof(string)` to get a floating point number from the user's text input instead of `stoi`. I could have used the `float powf(float, float)` function to square the number, but I used `num*num` instead since this is faster in an instance like this with a pre-determined integer power (in this case 2). Again I later came back and improved input handling by wrapping `stof` conversions in `try {} catch {}` blocks.

Task 4

I split my box-making code off into its own function for modularity; it takes two arguments: the text to put in the box, and the padding amount. The length of the string plus padding and box edges is stored as a local variable to simplify conditions in for loops later. The code then passes over multiple for loops to do things like generating a whole line of stars for the top, generate a line of spaces bounded by stars (and then output that a number of times based on the padding amount), and output the padding around the actual text. I used inline for loops for all of these, since each loop only contains a single `cout << something` statement, which considerably shortens the code and makes it easier to read. I generated the padding line (i.e. line of spaces with a star at each end) as a string and then repeatedly outputted that string, rather than building the padding line again each time, avoiding the use of nested for loops and making the code more efficient. I also discovered over the course of this challenge that `cin >>` supports automatically converting its result to an integer if the destination variable is an integer, which would slightly simplify some code in previous challenges. However I still used `getline` for the text itself so that the user can type multiple words without problems.

Task 5

My code is split into three functions, one for fully capitalising a string, one for full decapitalising it, and one for capitalising it on only the first letter of a sentence. The code primarily makes use of the standard library `char toupper(char)` and `char tolower(char)` functions to capitalise and decapitalise characters easily. The first two functions are simple `for` loops where we loop over the input and add the capitalised/decapitalised letter to the output buffer, before we return this buffer at the end. The latter function is more complex:

- We keep track of the current state of the text, with a local variable telling the program whether it is waiting to capitalise the first letter of a new sentence
- While this flag is set, we check to see if the current character is an alphabetical letter, by comparing it to capital and lowercase 'A' and 'Z' (since alphabetical letters in the ASCII character set are laid out, very sensibly, contiguously, though capital and lowercase are separated). Doing this with the `chr >= 'A' && chr <= 'Z'` condition format is great because it makes it easy to see what is actually being checked
- If the flag is set and the current character is an alphabetical letter, we must capitalise it, add it to the output buffer, and clear the end-of-sentence flag
- If the flag is set but there is no alphabetical letter (e.g. a space after a fullstop), we add the letter to the output buffer but don't clear the flag
- Otherwise, we just add the character to the output buffer, checking to see if the character we're adding is a fullstop, exclamation mark, or question mark, and if it is then we set the end-of-sentence flag

The main problem I encountered was that `cin >> text` only gets up to the next whitespace character and puts it in `text`, meaning that strings with spaces in (i.e. most sentences written by humans) would only get their first word read. This was fixed by using the `void getline(istream &, string)` function, which pulls an entire line up until it reads a newline character into the provided string variable.

Task 6

This task was made a lot easier by the fact that the random function is already written for us, so just calling it with appropriate arguments was simple. However, since the function defined in main.h is not documented, it required a little bit of experimentation to prove whether or not the upper and lower bounds were inclusive (through testing I found that they are both inclusive). We can use a simple `while (true) { }` loop with a `break` statement when the player guesses the number. Alternatively a conditional loop could be used, but this setup has the advantage that we don't need to duplicate code to prompt the user for input (i.e. we don't need to have an out-of-loop first prompt before the loop begins). Inside this `while` loop we prompt the user for a guess with `cout` and `cin`, check how far away from the secret number it is, break if its equal, or otherwise give the user a hint as to how warm they are:

- We first calculate the absolute difference from the secret number to the user guess, making the following `if` statements much simpler
- The next part was actually the trickiest part, since the description of the challenge talks about the player guessing **exactly 50 away** for the 'freezing' response for instance, and does not specify if the range of difference values covered by 'freezing' should therefore be 50-100 or 35-50. After somewhat clarifying with a tutor, I treated the challenge description as meaning **50 or more away** from the secret number (and following the same method for the other ranges)
- This did mean adjusting the 'boiling' range to also cover being 1 away from the secret number; previously I had an extra condition for being this close named 'super duper boiling'

- I used a chain of inline `if...else if...else` statements to implement these ranges in order to keep the code concise and the conditions on the `if` statements simple, and to avoid nesting

Each iteration of the `while` loop also increments the guess counter variable. When we escape the loop, the user is told what the number was and how many guesses they took to find it. I later added better input checking to this task, by way of using `getline` for the user input and adding a `try {} catch {}` block around the string-to-int conversion.

Task 7

I made use of a `const vector<string>` to store the character classes, since it's easy to get the length of this data type. I wrote a utility function which outputs a description of my character details struct. This could have been written as an overload of the `<<` operator. My character details struct contains two members, a name as a string, and a class ID as an integer (where the integer indexes the character class vector). When getting the user's player class, I used a simple `while (true) { }` loop, which prompts the user with the list of character classes each iteration, reads their input as an integer, and breaks only if the user's input was within the valid range. The user is also told if their input is valid or invalid, and which class they picked if they pick a valid option. If the user doesn't enter a number at all, a `try {} catch {}` block handles this and prompts the user for input again. When getting the user's character name I used `getline` in order to ensure the player could enter multi-word names (like maybe, 'Argoth the Destroyer', or whatever). I initially had problems with this however, since `getline` would return nothing (leading to a blank character name), skipping immediately due to it detecting the trailing newline from the previous user input. I resolved this by using `cin.ignore()` just before `getline`, to tell it to ignore the trailing newline (as suggested by answers on stack overflow). I used a pointer to my character details struct, so that a character could be passed easily into functions such as the `describe_character` function, and written back to by those functions potentially (I could imagine a function which might modify the character to upgrade their stats).

Task 8

My struct representing an inventory slot contains two fields: an item ID `int`, which indexes the `item_descriptions` vector (where -1 represents an empty slot), and an item description `string`. I again used a `const vector<string>` to store the list of item descriptions, as it's easily expandable and has a useful `.size()` method for performing bounds checks. I also developed a few utility functions:

- A function which just gets an item description from an ID (an `unsigned int`), and just contains a protective `if` statement to prevent indexing the item descriptions `vector` out of range
- A `split_string` function which takes in a string and a char, and will use the `string.find` and `string.substr` methods to split the input into a `vector<string>` at the delimiting character. This method is only used once but is very important to the command processing code and makes sense to have it abstracted as a separate function for clarity and modularity

The 'inventory' itself is managed by a class. This allows us to hide the actual backing data for the inventory, a `vector<inventory_slot>`, which can be protected by only allowing access through methods (the backing data member is made private). These methods allow for setting the contents of slots to particular item IDs, clearing slots, getting the contents of an inventory slot, setting the size of the inventory, and getting the size. Each of these methods has the appropriate validation, including checking for indexing the inventory out of range The user is prompted for a positive integer to initialise the inventory size, in a loop which repeats until a valid input is received. The only new thing to note here is handling exceptions from the `stoi` function using a `try...catch` block. If this function throws an error, we should simply prompt the user for another input. After initialising the inventory, we start a main program loop in which commands are processed, this is a `while` loop predicated on the `continue_mainloop` variable which allows us to break out from inside nested loops. When getting the user's command input, I had some problems again with `getline` being tripped up by a trailing newline, but having multiple inputs previously seemed to cause unpredictable behaviour. I solved this by repeatedly calling `getline` until it manages to get some input (i.e. a `string` of length > 0) without actually re-prompting the user. After the string is split at the space character, the only thing left is a lot of `if` statements to perform validation and branch depending upon the command entered, and the arguments passed to the command. Depending on the command, we must check the number of arguments provided (by looking at the number of elements in the output of `split_command`), and check that the numerical values entered are valid and in range for what they're indexing (e.g. the inventory or the list of items). Whenever an error condition is encountered, such as a 'set' command not having enough arguments, the `continue` keyword is used to skip back to the top of the loop and ask the user for a fresh command. I tested my input validation with a variety of different inputs, (valid, invalid, and edge case), so I'm confident that it would catch all invalid inputs. Although I didn't have any real problems with this challenge besides the `getline` shenanigans, it was a more interesting challenge than previous ones. I later added more input handling to prevent exceptions, specifically adding `try {} catch {}` blocks around calls to `stoi` and re-prompting the user for input if an exception occurs.

Task 9

The `Vector2` class itself is very simple, with `float x, y` variables. However, for some OOP practice I embedded `magnitude` and `distance` methods in the class, as well as a constructor which takes two `floats` for x and y. This `magnitude` method does exactly what it sounds like, and its implementation uses the same approach as earlier of `x * x` instead of `powf(x, 2)` for a slight speed boost, and the `sqrt` function to complete Euclid's formula. The distance method takes another `Vector2` as input and subtracts the self `Vector2` from it, before returning the magnitude of the resulting `Vector2`. In order to do this subtraction, I wrote a set of operator overloads accepting `Vector2` types: adding, subtracting, multiplying and dividing two `Vector2`s, and also multiplying and dividing a `Vector2` by a float. I also wrote a `<<` stream operator overload so that I could easily output my `Vector2` type to the console. However, these operators created a problem: they need the `Vector2` class to be declared, including x and y fields, meanwhile the class methods, mainly the `distance` method require the operators to already be declared (subtraction specifically), leading to a dependency loop. I solved this by declaring the class as a stub, then defining the operators, then finally implementing the methods from the `Vector2` class. Ideally these implementations would be done in a dedicated file (to form a pair of 'vector2.h' and 'vector2.cpp'). As instructed by the challenge description, I also wrote a `GetDistanceBetweenPoints` function which simply calls the `Vector2::distance` method and returns the result. The rest of the code back in main is just constructing a pair of `Vector2`s, calling `GetDistanceBetweenPoints` and outputting the result. This was an interesting challenge! I've had practice writing operators and mathematical classes before so this was a reminder for me.

Task 10

I came up with two `struct`s one for the marking information of each assessment, with the appropriate members based on reading the CRG. I had the idea of having the structs initialised with values equal to the maximum mark for those grades, meaning that to find the maximum marks for things (e.g. for finding percentage marks), I can just instantiate a new struct and read its default values. I wrote a few utility functions:

- One for adding up all the marks in the different challenges of the first criterion in assessment 1
- One for getting the total, weighted score for assessment 1, using the 70-30 split as described by the CRG
- One for getting the total, weighted score for assessment 2, using the 60-20-20 split as described by the CRG
- One which abstracts away the process of getting a validated integer input from the user. This gets used **`a lot`** in `main`, and since we want to do a little `while` loop to repeat until we get valid input, it makes sense to have this as a self-contained function. It supports having custom upper and lower bound limits and a specific prompt for the user
- One which returns a description for which CRG band a percentage score falls into, based on the ranges defined in the CRG

The rest of the program is lots of getting input from the user to populate the structs `a1g` and `a2g`. Once all this is done, we can calculate extra information like the total assessment 1 criterion 1 score, the total percentage score of assessment 1, the total percentage score of assessment 2, and the CRG grade bands for both of these and the individual criteria within. We can then output all of this information, nicely formatted, to the user. Again, I tested this thoroughly with various correct inputs (as well as checking the error checking with invalid inputs) and the output matched what was expected from what is written in the CRG.