Common Compilation Errors

This addresses common mistakes that cause code to compile locally but fail on Gradescope. It's important to understand these issues early on so you can avoid them in future projects.

Spelling Errors:

Function names are case-sensitive. If a function in the specifications is spelled with a capital letter, like `someConstructor`, but your code spells it as `someconstructor`, it will not compile on Gradescope.

Incorrect Parameter List:

The order of parameters in your function signatures must match exactly as specified. For example, if the specs require `someConstructor(const std::string& name, int level, bool enemy)`, but you write `someConstructor(const std::string& name, bool enemy, int level)`, it will not compile.

Missing Function Definitions:

All functions required by the specifications must be defined. If the specs ask for five functions but you only declare four, your code will not compile on Gradescope. You can use stubs (dummy implementations) for functions if you're still working on their implementation.

Default Parameters:

If a function in the specifications includes a default parameter, such as `someConstructor(const std::string& name, int level, bool enemy = false)`, omitting the default value in your implementation will cause compilation to fail.

Missing Includes:

Gradescope requires explicit includes for all necessary headers. For example, if your code uses `std::reverse()` from the `<algorithm>` header, you must include `#include <algorithm>` at the top of your file.

Incorrect Handling of Const References:

When a parameter is expected to be passed by const reference, such as `someConstructor(const std::string& name, int level, bool enemy)`, removing the `const` or the reference (`&`) will cause the code to fail compilation.

Missing Const Functions:

Getter functions that do not modify the class should be declared as const. For example, `std::string getName() const` is correct, while `std::string getName()` without `const` will not compile if the specifications expect the former.

Incorrect File Names or Class Names:

Make sure you submit files with the exact names specified in the project requirements. If the specs expect `Dish.cpp` but you submit `dish.cpp`, it will not compile. The same applies to class names.

If you encounter the “does not compile on Gradescope” error, review these common issues. Ensure your code adheres strictly to the specifications, and test locally before submitting. Remember, once your code compiles on Gradescope, it will be tested for correctness. If you're certain that your code is compliant with everything mentioned above, then you should also test your code using a lab machine (which runs the same Linux version as the autograder). You can do this in person at HN1001B or remotely. The instructions for connecting remotely can be found on Blackboard or in a shortened version below:

Connecting Remotely

This is a step-by-step guide to using your terminal to run code on the Linux lab machines (which use the same environment as Gradescope). This is essential for debugging your code and you should do it before submitting (and before asking any questions about compilation errors on Gradescope).

This process seems intimidating at first, but it’s very easy, especially once you’ve done it once! Let's take it step-by-step and try to cover everything.

Requirements:

- **For Mac users:** Macs should have Xcode installed by default. If not, install Xcode or at least the command line tools.

- **For Windows users:** Follow this tutorial to install the Windows Subsystem for Linux (WSL), which is necessary for terminal access and compiling: https://okunhardt.github.io/documents/Installing_WSL.pdf
  - If you're unable to use WSL, you can also use Putty and Xming (instructions are on blackboard).
- **For Linux users:** Ensure you have a terminal set up with **ssh** and **git** installed. Typically, most Linux distributions already have what you need. You can verify this by running:

  ```
  ssh -V
  git --version
  ```

  If **ssh** or **git** is missing, install them with:

  ```
  sudo apt-get install openssh-client
  sudo apt-get install git
  ```

Step 1 (Preferred Method – Using Git to Clone the Repo)

- On your local machine, you should have a separate folder where your GitHub repository is cloned.
- Open the terminal, and **ssh** into the Hunter servers with the following command:

  ```
  ssh john.smith99@eniac.cs.hunter.cuny.edu
  ```

  - **Replace john.smith99 with your own username, which should be the same as your Hunter/CUNY email before the @ symbol.**
  - (Input your password, it won't display any characters, then press enter.)
  - It may ask if you trust this server; type "yes."
- Once logged in, **ssh** into one of the lab machines using:

  ```
  ssh cslab10
  ```

  - If machine 10 isn't available, you can replace "10" with any number from 1-20.
  - (Input your password again, it won't display any characters, then press enter.)
  - It may ask again if you trust this server; type "yes."

- Navigate to a directory where you want to store the repository by using the `cd` command, or create a new one using `mkdir`:

  ```
  mkdir new_project_folder
  cd new_project_folder
  ```

- Clone your GitHub repository to the server by using the following command:

  ```
  git clone https://github.com/YourRepo/YourProject.git
  ```

  - Replace the URL with your actual GitHub repository URL.
- This will copy your project files to the lab machine. You can now proceed to compile and run your code as described below.

Step 2 (Compile & Run Remotely)

- After cloning the repository, navigate into your project folder:

  ```
  cd YourProject
  ```

- Use the `make` command to compile your files:

  ```
  make
  ```

  - This will compile your program and print any compilation errors or warnings.
- If your code compiles successfully, run the executable:

  ```
  ./main
  ```

  - This will execute your program and display the output in the terminal.
- When finished, use the `exit` command to end the session:

  ```
  exit
  ```

Alternative Method (Not Preferred – Using SFTP)

If you prefer not to use Git, you can still manually transfer your files using SFTP. However, this method is less efficient than cloning the repository directly.

Step 1 (Local Folder)

- On your local machine, ensure you have a dedicated folder containing all your program files. This folder should not contain any unnecessary files.
- Open the terminal in that folder (right-click and choose "Open Terminal" or use the **cd** command to navigate to it).
- Run the **ls** command to verify the contents of the folder:

  ```
  ls
  ```

  - This will list all files in the folder, confirming that your program files are present.

Step 2 (Connect Remotely For Transfer)

- Use the following command to initiate a secure file transfer (SFTP) session with the Hunter servers:

  ```
  sftp john.smith99@eniac.cs.hunter.cuny.edu
  ```

  - **Replace john.smith99 with your own username, which should be the same as your Hunter/CUNY email before the @ symbol.**
  - (Input your password, it won't display any characters, then press enter.)
  - It may ask you if you trust this server; type "yes."
- Once logged in, use the **ls** command to list the directories on the server:

  ```
  ls
  ```

  - Navigate to the desired folder on the server using the **cd** command, or create a new directory with **mkdir**:

  ```
  mkdir new_project_folder
  cd new_project_folder
  ```

Step 3 (Transfer Files)

- Now that you're in the desired destination on the lab machine and your local terminal is still in the directory with your project files, use the following command to transfer all your files:

  ```
  mput *
  ```

  - This will upload all files from your local folder to the directory on the lab machine.
- After the transfer is complete, use the `exit` command to leave the SFTP session:

  ```
  exit
  ```

- You can now proceed with the **Compile & Run Remotely** steps outlined above.

Authors: Michael Russo, Arsen Tumanian, Prof. Wole