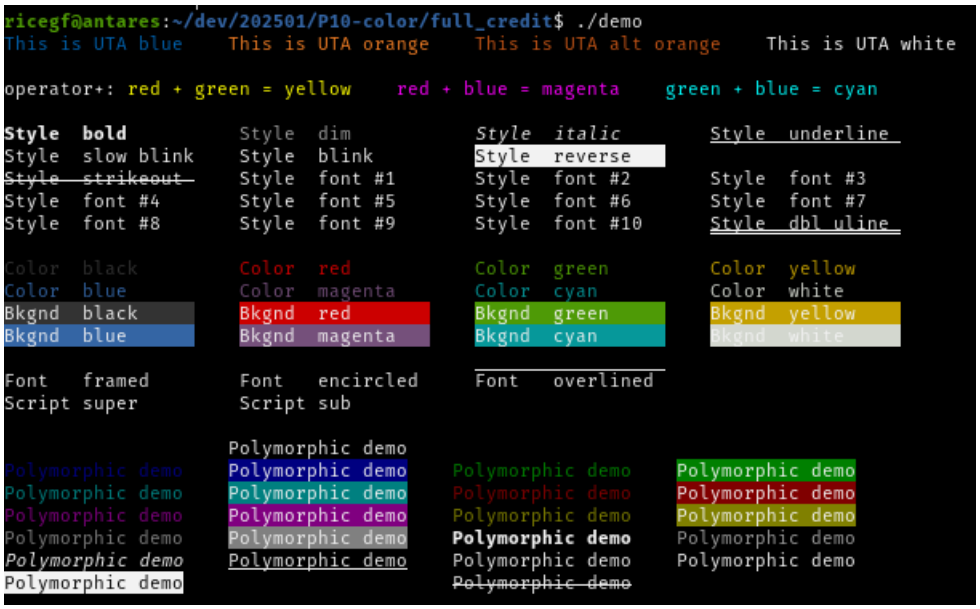
# P10 - Coding in Color

## Due Tuesday, April 15 at 8 a.m.

CSE 1325 - Spring 2025 - Homework #10 - Rev 0

# Assignment Overview

Let's code with all the colors of the wind, exercising such C++ skills as inheritance, `std::map`, and operator overloading including the spaceship <=>. We'll create objects that behave as I/O manipulators to stream out colorful, stylized terminal text like this!




**IMPORTANT:** To view the output, this assignment requires an "ANSI-compatible" terminal. Most Linux terminals including CSE-VM's terminal, the Mac default terminal, the git bash and minTTY terminals on Windows, and the VS Code terminal (including on GitHub Codespaces) are all likely to implement at least some colors and many of the fonts. The COMMAND.EXE (DOS) shell on Windows may not work - if not, switch to a compatible terminal when running your code.

The tests at cse1325-prof/P10/baseline will let you check your terminal, regression test your code, and recreate the above output. These may change, so check back periodically.

- **test_terminal** will check your terminal for ANSI support. Use `make test_terminal ; ./test_terminal` to run it.
- **test_font** will test your `Font` class. Use `make test_font ; ../test_font` to run it.
- **test_color** will test your `Color_mode` and `Color` classes. Use `make test_color ; ../test_color` to run it for the Full Credit level, and `make test_color ; ../test_color bonus` to run it for the Bonus level.
- **demo** will generate the above as best your terminal supports. Use `make demo ; ./demo` to run it.

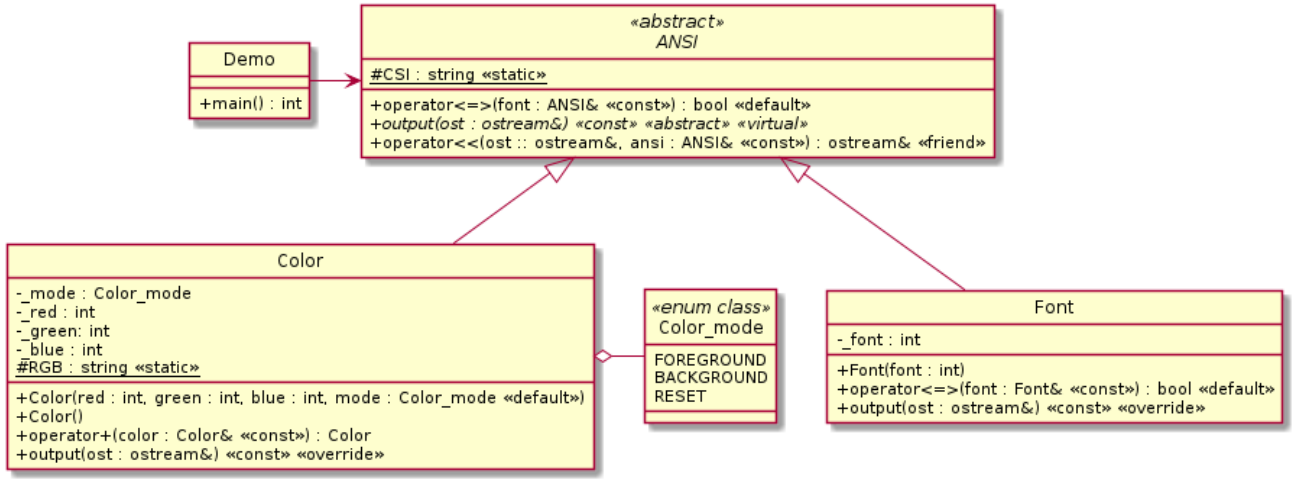
For those interested, more on ANSI terminal commands may be found here.

https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797
https://en.wikipedia.org/wiki/ANSI_escape_code

# Full Credit

Test code to help you verify correct class implementation, as well as demo.cpp containing main used to produce the above output, is provided at cse1325-prof/P10/baseline. You are NOT required to add this code to your own repository, although it is fine if you do. The suggested solution is just over 100 lines of code total.

Consider the following class diagram, which specifies the `ANSI` class and its `Font` and `Color` subclasses.




# Program Overview

Declare each of the classes and the enum class in a corresponding .h file, then implement in a .cpp file. A Makefile is provided below. Demo and test code is provided at cse1325-prof/P10/baseline.

Superclass `ANSI` defines the Control Sequence Introducer (CSI), which is an ESC and [ character pair, along with overloading `operator<<` via a friend function which will call subclass overrides of abstract method `output`. (This is necessary because functions such as `operator<<` can't be called polymorphically, but methods such as `output` can. Recall that polymorphism in C++ requires calling the method using a pointer *or reference*, and the `ansi` parameter is a const reference.)

Subclass `Font` overrides `output` to command the terminal to select a font specified by the constructor parameter. The characters output are the CSI pair, the text version of the `_font` integer, and `'m'`.

Enum **class** `Color_mode` enumerates `FOREGROUND` (set the color of the text itself), `BACKGROUND` (highlight the text background with color), and `RESET` (clear all color and font settings in the terminal). Also overload the `operator<<` *function* for `Color_mode` to stream lowercase versions of the enumerations.

Subclass `Color` overrides `output` to command the terminal to select a foreground (text) or background (highlight) RGB color. The default for `mode` is `Color_mode::FOREGROUND`. The default constructor selects `Color_mode::RESET` and ignores the other fields. RBG is ";2;" to select the RGB color terminal command.

> The `RESET` code is the same as for `Font{0}`. The foreground code is the CSI pair followed by `38;2;r;g;bm` (NO whitespace) where `r`, `g`, and `b` are integers (as decimal text) from 0 to 255 (throw `std::invalid_argument` if a `red`, `green`, or `blue` parameter is not within this range). Similarly, the background code is the CSI pair followed by `48;2;r;g;bm`. Larger `r`, `g`, and `b` integers represent more of the corresponding red, green, and blue color components.
>
> `Color`'s overloaded `operator+` combines the current object's and parameter's `_red`, `_green`, and `_blue` color components into a new `Color` object to return. Combine each color component independently using the formula `255-(255-a)*(255-b)/255`. a is this object, b is the parameter.

Exceptionally detailed guidance follows for those who need to follow it carefully or to use only as needed.

## Abstract Class ANSI

First, write the abstract `ANSI` class declaration in file ansi.h.

- Write a guard.
- Include `<iostream>`
- Declare the class. Recall that abstract classes in C++ are declared just like non-abstract classes.
- Declare the spaceship `operator<=>` as default. (If your compiler won't support the spaceship, instead declare the 6 comparison operators as described in Lecture 21. You don't actually need a `compare` method here, because class `ANSI` has no non-static fields to compare! Just hard-code the inline definitions as `true` for ==, >=, and <= and `false` for `!=`, >, and <. Note that if a subclass declares a spaceship, as `Font` does, its superclasses will need the comparison operators defined as well!)
- Declare `operator<<` (Lecture 21) *as a friend* of class `ANSI`. The promised function signature is:

```
std::ostream& operator<<(std::ostream& ost, const ANSI& ansi);
```

- Declare abstract const method `output`. Include the keyword `const` right after the parameter list to promise `output` won't modify the object. Recall that "abstract`` in C++ is specified with `= 0;`.
- Declare the constant `CSI` as a static class member. Recall that C++ static fields must be *declared* in the .h file and then *defined* in the .cpp file to allocate memory.

Next, implement the friend function and static field of `ANSI` in file ansi.cpp.

- Include the header file `<ansi.h>`.
- Write friend function `operator<<`.
    - Do NOT reuse the keyword `friend` - that's only for the .h file.
    - Do NOT put `ANSI::` before `operator<<` - this is a *function*, not a method!
    - The implementation should polymorphically call `ansi.output` passing the `ost` parameter and then return `ost`. Recall that since `ansi` is a *reference*, polymorphism will work - no need to convert it to a pointer! Polymorphism fails in C++ only on a *value* object.
- Also define const string `CSI` as an escape and open square bracket. The C++ way to do this is to use the exact string `"\033["`. The `\033` is ESC (in octal) and `[` is the open square bracket char.

Verify that this class compiles to file ansi.o.

- If you're using the gcc compiler, the command is `gcc --std=c++20 -c ansi.cpp`.
- If you're already using the Makefile below, good for you! The command is `make ansi.o`.
- You'll need to work out the command if you're using a different compiler.

## Class Font

Next declare class `Font` in file font.h.

- Write a guard.
- Include `"ansi.h"` and, if you like, `<iostream>`. The latter isn't strictly necessary, since ansi.h also includes it, but explicit is often better than implicit. `iostream`'s guard will ensure it's only compiled once.
- Declare class `Font` as a subclass of `ANSI`.
- Declare a constructor with parameter `int font` which has a default value of 0. This is also the default parameter since all of its parameters have default values, right?
- Declare method `output` to override the superclass method, and ask the compiler to verify this using the `override` keyword just before the semicolon. Add `const` AFTER the closing parameter list parenthesis, NOT before the return type (which would make the *return type* const rather than the *method*).
- Declare the spaceship `operator<=>` as default. (If your compiler won't support the spaceship, instead declare the 6 comparison operators and supporting code as described in Lecture 21. Or use the preprocessor variable to support both!)
- Declare private int field `_font`.

Next implement `Font`'s constructor and method in file font.cpp.

- Include the header file `<font.h>`.
- Implement the constructor. Do NOT include the default parameter value - that's for the .h file only! Initialize the field from the parameter *using an init list*. No data validation is required.
- Override method `output`. The `const` is also required here, but `override` is omitted. To change the font, first stream out `CSI`, then `_font`, and then `m` **with no intervening whitespace**. This switches the terminal to using the font number in `_font`.

That's it! Test-compile your code as described in **class ANSI** above, for example `make font.o`. Once compiled, you should be able to test using the code provided at cse1325-prof/P10/baseline.

## Enum Color_mode

Most terminals can apply the color to the foreground (that is, the characters themselves) or to the background (similar to highlighting on a page). We can also reset to the default foreground and background color.

Therefore, in file color_mode.h, declare `enum Color_mode`

- Write a guard.
- Include `<iostream>`
- Declare the enum *class* with the enumeration values from the class diagram.
- Declare the `operator<<` function for the `Color_mode` type. (It is NOT a friend, since it's not declared within the enum itself.) The promised function signature is:

```
std::ostream& operator<<(std::ostream& ost, const Color_mode& cm);
```

Then, in file color_mode.cpp, define `operator<<` *using a map*.

- Include `"color_mode.h"` and `<map>`.
- Define a `std::map` named `text` with type `Color_mode` as key type and `std::string` as value type. Initialize it to map the 3 enumerations to "foreground", "background", and "reset", respectively.
- Implement `operator<<` to stream `text[cm]` or `text.at(cm)` (your choice - do you know the difference?) to `ost`, and then as always return `ost`.

Test-compile your code as described in **class ANSI** above, for example `make color_mode.o`. Once compiled, you should be able to test using the code provided at cse1325-prof/P10/baseline.

## Class Color

Next declare class `Color` in file color.h.

- Write a guard.
- Include `"ansi.h"`, `"color_mode.h"`, and, if you like, `<iostream>`.
- Declare class `Color` as a subclass of `ANSI`.
- Declare a default constructor. (This will be the "reset" command to clear all colors and fonts.)
- Declare a non-default constructor with `red`, `green`, and `blue` int parameters and `Color_mode mode` with default value `Color_mode::FOREGROUND`.
- Declare method `operator+` for adding 2 colors.
- Declare method `output` overrides the superclass method, and ask the compiler to verify this using the `override` keyword. Also declare that this method is itself `const`.
- Declare protected constant `RGB` as static. The will be the special ANSI command to define a new 24-bit color.
- Declare private `Color_mode` field `_mode` *first*, then private int fields `_red`, `_green`, and `_blue`.

Next implement `Color`'s constructors, methods, and constant field in file color.cpp.

- Include the header file `"color.h"` and `<map>`.
- Define a `std::map` named `code` with type `Color_mode` as key type and `int` as value type. Initialize it to map the 3 enumerations to 38, 48, and 0, respectively. These are the ANSI "color" numeric commands for setting the color of characters, highlighting, and resetting back to defaults.
- Implement the default constructor, which simply sets field `_mode` to `Color_mode::RESET` in the init list.
- Implement the non-default constructor, which initializes all 4 fields *using an init list*. No data validation is required.
    - Initialize `_red`, `green`, and `_blue` to each corresponding parameter using an init list. In the body, if any of these are not within [0,255] inclusive, throw a `std::illegal_argument` exception.
    - Initialize `_mode` to parameter `mode`.
- Implement method `operator+`. You will calculate each new color component red, green, and blue separately from the corresponding values of `this` (a in the formula) and the parameter (b) using the formula `255-(255-a)*(255-b)/255`. You MAY write this as a function or private method if you like for DRY purposes. Return a `Color` instance using the calculated values.

  Other options for combining each respective color component I've seen include averaging them (but this gives very dark colors) or taking the square root of the sum of the squares (but this tends to be so bright the color components sometimes exceed 255).

- Implement method `output`. The `const` method declaration is also required here.
    - First stream out `CSI` and the mode code from the `std::map code`. Do NOT include any newlines or spaces!
    - Next, if `_mode` is not `RESET`, stream out `RGB`.
    - Next, if `_mode` is not `RESET`, stream out the 3 colors `_red`, `_green`, and `_blue` separated by semicolons. Do NOT include any newlines or spaces!
    - Finally, stream out `m`.
- Finally, define the const static `RGB` field as ";2;".

That's it! Test-compile your code as described in **class ANSI** above, for example, `make color.o`. Once compiled, you should be able to test using the code provided at cse1325-prof/P10/baseline. The demo code should also work for you now!

Time to move on to the Bonus.

## Makefile

Here's a Makefile that should work for all levels of this assignment. Remember that the first character of indented lines must be a TAB, not a SPACE!

```
CXXFLAGS = --std=c++20

demo: demo.o ansi.o color.o font.o *.h
	$(CXX) $(CXXFLAGS) demo.o ansi.o color.o font.o -o demo
	@printf "Now type ./demo to execute the result\n\n"

demo.o: demo.cpp
	$(CXX) $(CXXFLAGS) -c demo.cpp -o demo.o

ansi.o: ansi.cpp
	$(CXX) $(CXXFLAGS) -c ansi.cpp -o ansi.o

font.o: font.cpp
	$(CXX) $(CXXFLAGS) -c font.cpp -o font.o

color: color.cpp
	$(CXX) $(CXXFLAGS) -c color.cpp -o color.o

color_mode.o: color_mode.cpp
	$(CXX) $(CXXFLAGS) -c color_mode.cpp -o color_mode.o

clean:
	rm -f *.o *.gch a.out demo
```

# Bonus

Write the comparison operators for class `Color` as well. You can't use the spaceship here, because the rules are more subtle.

- If `_mode` in both objects is `Color_mode::RESET`, the objects are equal regardless of the other fields.
- Otherwise, the comparison precedence is `_mode`, `_red`, `_green`, and `_blue`.

Use the "bonus" command line argument to also test your `Color` comparisons.

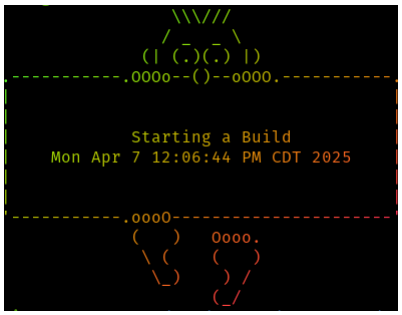
It would make more sense to compare luminence or brightness rather than individual fields here. Feel free to do that if you like, and modify test_color.cpp to match the better definition. I just ran out of time. The commonly accepted formula is

Brightness = 0.2126 * _red + 0.7152 * _green + 0.0722 * _blue;

# Extreme Bonus

Use the `ANSI` class and subclasses to write your own demo or game. Here are a few ideas to get you started.

- Write an I/O manipulator class that streams out a random `Color` (and maybe `Font`) code each time it is invoked. Keep the red, green, and blue values above 128 to ensure the color is easy to see on a dark background terminal. Also include a `Reset` manipulator that resets the terminal mode (using `Color()` or `Font()`), since the terminal probably will NOT reset itself when your program exits!
- Write an I/O manipulator class that creates colorful text for the string given in the constructor. Keep the red, green, and blue values between 128 and 255, but cycle them using different prime numbers such as 19, 37, and 53. Set the color of each succeeding character to the next color triplet. The `lolcat` open source program does this *very* effectively!




- The ANSI Code links in the intro also define cursor controls. Write a `Cursor` class that includes methods for manipulating the cursor with C++ string. This actually exists in the open source world as the famous "curses" family of libraries.

https://en.wikipedia.org/wiki/Curses_(programming_library)

- Create a `Card` class with two fields:
    - An int or enum rank from 1 (A) to 10 and 11 (J), 12 (Q), and 13 (K)
    - An enum suit including spades (♠), hearts (♥), diamonds (♦), and clubs (♣).

  When you stream out a heart (♥) or diamond (♦) `Card`, make it red using the `Color` code (`Color red{255,0,0};`) or `Font` code (`Font red{31};`). Write a `main` function that prints a few `Card` instances for each suit to demonstrate that it works.

Then write a `Deck` class using the `Card` class you may have created at the Bonus level, and write a game using your new `Deck` of cards.

Blackjack is a good choice, since the dealer must always follow simple "programmed" (ahem) moves while the player has choies based on the card count (double-ahem!).

https://en.wikipedia.org/wiki/Blackjack

Or write the classic children's game "War" ("Battle" to the British), where ALL of the moves are programmed! :D Yes, I've written this one, and discovered that a majority of games go into extended infinite loops (computers don't get bored and give up like children do). To avoid these, I added "General Random" who amazingly flips the battle 1 out of 100 times at random so that the *lower* card wins.

https://en.wikipedia.org/wiki/War_(card_game)