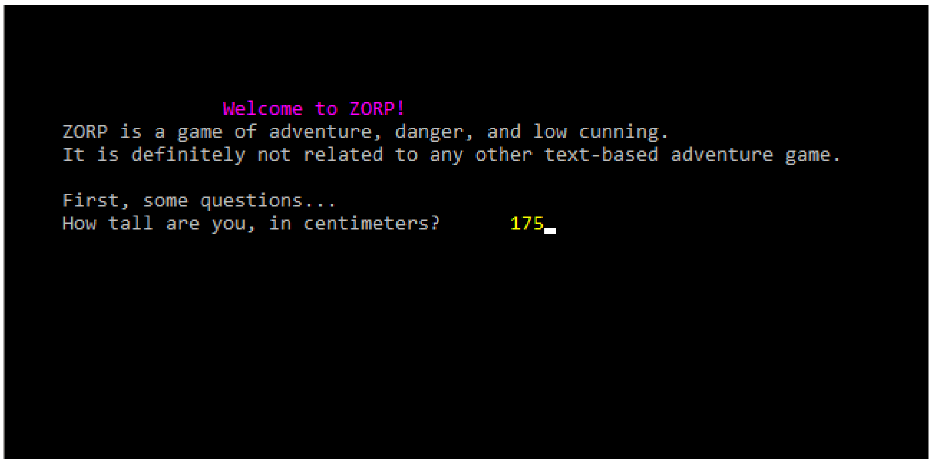
Tutorial – Constants: Zorp part 2



```
                Welcome to ZORP!
ZORP is a game of adventure, danger, and low cunning.
It is definitely not related to any other text-based adventure game.

First, some questions...
How tall are you, in centimeters?      175_
```

Introduction:

In this tutorial, we will use some constant definitions containing console formatting codes to format the text that is printed to the console. Using constant definitions for this will simplify our code, make it more readable, and more easily modifiable.

This tutorial continues on from the previous Zorp tutorial for the session on *Variables*. If you have not yet completed that tutorial, you will need to do that before attempting this one.

Clearing the Screen:

As you develop your console-based games, there will come a point when you'll find it convenient to clear the screen.

Many students are familiar with DOS, and using the *system()* function to invoke the DOS *cls* command is simple and straightforward.

```
system("cls");
```

If you want to see the effect of this command, update your game by placing this code somewhere in your program, or simply open a terminal window (from the Windows start menu, search for 'cmd') and type in "*cls*".

You may remember in the last tutorial we discussed a similar system command ("pause"). As with the *pause* command, the *cls* command suffers from one possibly fatal flaw – it permanently ties our program to the Windows platform.

While this is very likely not going to be an issue (ever?) for many of you, it would be nice to know that we've taken some steps to ensure our code *could* be compiled for another platform (you know, just in case that situation ever comes up). At the very least, I'm guessing many of you may be curious about the solution, even if you don't think you'll need it just yet.

On Linux and Unix platforms (commonly called *nix platforms), we can prompt the console to clear the screen via character escape codes written to the command line.

Essentially this allows us to clear the screen by outputting a very specific sequence of characters to *std::cout*.

```
        // clear the screen with an escape sequence
        std::cout << "\x1b[2J";
```

The above string, when written to the console, will clear the screen on *nix.

This sequence is usually combined with the command to move the cursor. This code will clear the screen and move the cursor to the top-left of the screen (replicating the functionality of calling *system("cls")*).

```
        // \x1b[2J clears screen, \x1b[H moves the cursor to top-left corner
        std::cout << "\x1b[2J";
```

Unfortunately, unless we're working with a version of Windows that precedes Win'98, this won't work in Windows until we enable the console to handle *Virtual Terminal Sequences*. This requires a few lines of code when our program launches.

Virtual Terminal Sequences:

In Windows, to be able to pass commands to the terminal to control the display of text, we need to enable virtual terminal sequences.

Insert the following code just after the declaration of the *main()* function:

```
#include <iostream>
#include <windows.h>

void main() {
        // Set output mode to handle virtual terminal sequences
        DWORD dwMode = 0;
        HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
        GetConsoleMode(hOut, &dwMode);
        dwMode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;
        SetConsoleMode(hOut, dwMode);
    ...
}
```

The breakdown of this code line by is:

1. Create a variable to store the current console mode (*dwMode*)
2. Get the handle of the standard output (i.e., *cout*). A handle is akin to an identifier.
3. Using the handle, get the console window's current output mode.
4. Amend the current output mode to also include the 'enable virtual terminal processing' flag. We want to amend the current state (rather than using this flag directly) because the console may have other settings that we wish to keep.
5. Set the console output mode. After calling this function our console window will be enabled to process our text formatting escape sequences.

Now that we've enabled the virtual terminal processing mode, we can clear the screen by doing the following:

```
#include <iostream>
#include <windows.h>

void main() {
	// Set output mode to handle virtual terminal sequences
	DWORD dwMode = 0;
	HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
	GetConsoleMode(hOut, &dwMode);
	dwMode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;
	SetConsoleMode(hOut, dwMode);

	std::cout << "Hello world!";
	std::cout << "\x1b[2J\x1b[H";
	std::cout << "Did you see it?";
	return;
}
```

(Don't put this code in your *Zorp* game, it's just for illustration.)

What's the point? I mean, cls worked fine, was one line of code, and was easy to remember, right?

Well, now that we have this command mode enabled we've actually unlocked A TONNE of functionality we can use to make our game a lot more interesting (did you see the colours in the screen shot at the beginning of this tutorial?)

This Microsoft reference page lists all the commands we can pass the terminal:
https://msdn.microsoft.com/en-us/library/windows/desktop/mt638032(v=vs.85).aspx#example

Here are the highlights:

- Move the cursor to any screen coordinate
- Change the background and/or foreground colour of text
- Delete individual lines, or characters
- Save and restore the position of the cursor
- Draw ASCII line drawings
- Set a scroll region

Along with a few more other things.

We've unlocked a range of commands that allow us to make our text adventure game visually interesting. And what's even better is that these commands will work on other platforms too.

Defining Pre-set Commands:

These commands are relatively easy to use. All we need to do is write the correct character sequence to the console.

Unfortunately, the character sequences themselves are a bit cryptic. This is where constant values can help.

Define the following constants values in your game:

```
#include <iostream>
#include <windows.h>

const char* CSI = "\x1b[";
const char* TITLE = "\x1b[5;20H";
const char* INDENT = "\x1b[5C";
const char* YELLOW = "\x1b[93m";
const char* MAGENTA = "\x1b[95m";
const char* RESET_COLOR = "\x1b[0m";
const char* SAVE_CURSOR_POS = "\x1b[s";
const char* RESTORE_CURSOR_POS = "\x1b[u";

void main() {
   ...
```

Each command is defined as a constant character array (i.e., a string). Strings and arrays are covered soon, in case the *char** syntax is unfamiliar to you.

The commands we've specified are:

- CSI – this acronym stands for Control Sequence Introducer. This character sequence is at the start of *every* command we'll pass to the console. It's defined here for situations where we can't use a pre-defined command.
- TITLE – this command moves our cursor to the x,y coordinate (20, 5). This is where we'll write the game's welcome title.
- INDENT – moves the cursor right by 5 characters. We'll use this to specify an indent when outputting text.
- YELLOW – sets the text colour to yellow.
- MAGENTA – sets the text colour to magenta.
- RESET_COLOR – resets the text colour to the default colour.

- SAVE_CURSOR_POS – save's the cursor's current position to memory.
  The position isn't stored in a variable that we have access to, its stored by the console itself.
  This means we can only save one position.
- RESTORE_CURSOR_POS – set the cursor to the last saved position.

Now, using these console commands will be as simple as including the const variable in a call to *std::cout*. For example:

```
std::cout << TITLE << MAGENTA << "Welcome to ZORP!" << RESET_COLOR << std::endl;
```

Before you go any further, take a quick look at the Microsoft refence page and check the description for each command we've created a *const* variable for.

https://msdn.microsoft.com/en-us/library/windows/desktop/mt638032(v=vs.85).aspx#example

If you like, you can create new *const* variables for commands you may want to use in your game.

Updating the Game's Formatting:

Now for the fun part; updating our game using these command constants to make our game look and feel a bit more exciting.

We're not going to add any new game logic. All we'll do is use the commands we've defined to control how our game looks.

Below is the full game update. As you make the changes to your game, think about what commands are being sent to the console and the effect it will have.

```
#include <iostream>
#include <windows.h>

const char* ESC = "\x1b";
const char* CSI = "\x1b[";

const char* TITLE = "\x1b[5;20H";
const char* INDENT = "\x1b[5C";
const char* YELLOW = "\x1b[93m";
const char* MAGENTA = "\x1b[95m";
const char* RESET_COLOR = "\x1b[0m";
const char* SAVE_CURSOR_POS = "\x1b[s";
const char* RESTORE_CURSOR_POS = "\x1b[u";

void main() {
	// Set output mode to handle virtual terminal sequences
	DWORD dwMode = 0;
	HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
	GetConsoleMode(hOut, &dwMode);
	dwMode |= ENABLE_VIRTUAL_TERMINAL_PROCESSING;
	SetConsoleMode(hOut, dwMode);
```

```
        int height = 0;
        char firstLetterOfName = 0;
        int avatarHP = 0;

        std::cout << TITLE << MAGENTA << "Welcome to ZORP!" << RESET_COLOR
                << std::endl;
        std::cout << INDENT << "ZORP is a game of adventure, danger, and low
                cunning." << std::endl;
        std::cout << INDENT << "It is definitely not related to any other text-based
                adventure game." << std::endl << std::endl;

        std::cout << INDENT << "First, some questions..." << std::endl;

        // save cursor position
        std::cout << SAVE_CURSOR_POS;
        std::cout << INDENT << "How tall are you, in centimeters? " << INDENT
                << YELLOW;

        std::cin >> height;
        std::cout << RESET_COLOR << std::endl;

        if (std::cin.fail()) {
                std::cout << INDENT << "You have failed the first challenge and are
                eaten by a grue." << std::endl;
        }
        else {
                std::cout << INDENT << "You entered " << height << std::endl;
        }

        // clear input buffer
        std::cin.clear();
        std::cin.ignore(std::cin.rdbuf()->in_avail());
        std::cin.get();

        // move the cursor to the start of the 1st question
        std::cout << RESTORE_CURSOR_POS;
        // delete the next 4 lines of text
        std::cout << CSI << "4M";

        std::cout << INDENT << "What is the first letter of your name? " << INDENT
                << YELLOW;

        std::cin >> firstLetterOfName;
        std::cout << RESET_COLOR << std::endl;

        if (std::cin.fail() || isalpha(firstLetterOfName) == false) {
                std::cout << INDENT << "You have failed the second challenge and are
                eaten by a grue." << std::endl;
        }
        else {
                std::cout << INDENT << "You entered " << firstLetterOfName
                << std::endl;
        }
        std::cin.clear();
        std::cin.ignore(std::cin.rdbuf()->in_avail());
        std::cin.get();

        // move the cursor to the start of the 1st question
        std::cout << RESTORE_CURSOR_POS;
        std::cout << CSI << "A";            // cursor up 1
        std::cout << CSI << "4M";           // delete the next 4 lines of text
        if (firstLetterOfName != 0) {
                avatarHP = (float)height / (firstLetterOfName * 0.02f);
        }
        else {
                avatarHP = 0;
        }

        std::cout << INDENT << "Using a complex deterministic algorithm, it has been
                        calculated that you have " << avatarHP << " hit point(s)."
                        << std::endl;


        std::cout << std::endl << INDENT << "Press 'Enter' to exit the program."
                        << std::endl;
        std::cin.get();
}
```

The funny thing is, now that we have all this functionality to control and manipulate text output, we’re not even using the command to clear the whole screen (which is what we started out wanting to do). Instead we’re deleting only the parts that we want to change.

Now that you have a little knowledge, experiment with the different command sequences listed on Microsoft’s site and see what you can come up with.