Il C, la console testuale e i colori

Gli **artisti** possono colorare il **cielo** di **rosso** perché **sanno** che è **blu**. Quelli di noi che **non sono artisti** devono **colorare** le cose come **realmente** sono o la gente **penserebbe** che sono **stupidi**.

Jules Feiffer

La console testuale, la nostra fidata cmd.exe o il nostro terminale Linux o macOS, sono notoriamente dei mondi spartani e minimalisti. Tuttavia, anche in questi contesti un tocco di colore può aiutare a rendere l'output maggiormente intuitivo e piacevole da gestire. Il problema è che per realizzare anche il più piccolo inserimento di colori, utilizzando il nostro codice C, dobbiamo faticare non poco a causa della eterogeneità dei vari sistemi operativi e ambienti di sviluppo.

Le sequenze di escape

Un primo approccio che possiamo provare a sfruttare è quello di pilotare il nostro terminale attraverso specifiche sequenze di comandi note come **sequenze di escape**. Più precisamente, si tratta di uno standard ANSI (ovvero dell'American National Standards Institute) che consente di inviare al terminale una serie di comandi, di norma utilizzando il carattere `Esc` seguito dalla parentesi quadra '`[`', con i quali controllare non solo i colori ma anche la posizione del cursore sullo schermo. Storicamente, si tratta di uno stratagemma molto datato che fu introdotto addirittura a partire dagli anni 70 e particolarmente 80 del secolo scorso per sostituire il vecchio metodo che consisteva nell'utilizzare comandi legati all'hardware dei singoli e specifici dispositivi. Nonostante siano passati moltissimi anni, questo metodo è ancora utilizzabile con una certa efficienza grazie al fatto che è possibile inviare comandi al terminale utilizzando caratteri ASCII standard.

Figura - Il terminale VT 100 della Digital (https://commons.wikimedia.org/wiki/File:DEC_VT100_terminal.jpg).

Può essere interessante sapere che uno dei primi e più popolari terminali a supportare questo nuovo standard fu il VT100 della Digital. Personalmente, lo ricordo ancora con un po' di malinconia in quanto

questo fu il primo terminale con il quale mi imbattei all'Università degli Studi di Salerno nella seconda metà degli anni 80.

Unix sì, Windows "nì"

Il primo problema che riscontriamo con l'utilizzo delle sequenze di escape è che esse sono supportate in maniera pressoché nativa solo in ambiente Unix like e non sotto Windows. Cominciamo comunque a verificarne il loro utilizzo almeno sotto il contesto Linux per prendere confidenza con questo strumento.
Un primo stralcio di codice minimale con il quale fare una veloce prova può essere il seguente:

```
#include <stdio.h>

int main ()
{
    printf("\x1b[32mUn mondo a colori\n");
}
```

Come si intuisce il contenuto da analizzare è il seguente:

```
"\x1b[32mUn mondo a colori\n"
```

In tale contenuto riconosciamo facilmente che prima della stringa "`Un mondo a colori`" è presente una arcana sequenza di simboli:

```
"\x1b[32m"
```

Ebbene, in realtà non è nulla di particolarmente complesso. Infatti, come già detto, le sequenze di escape iniziano con il codice del carattere `Esc` seguito da una parentesi quadra aperta. Tale carattere di escape corrisponde al valore decimale 27 che, in esadecimale, corrisponde a `1b` ed è proprio tale valore che inviamo al terminale usando la sequenza di due simboli "`\x`".
In successione troviamo il valore 32 che corrisponde al colore verde e infine la lettera `m` che serve a segnalare al terminale che stiamo inviando un comando di gestione di tipo grafico.
Se compiliamo il file sotto Windows e proviamo a lanciarlo nella console cmd.exe otterremo un deludente risultato simile a quanto riportato in figura:

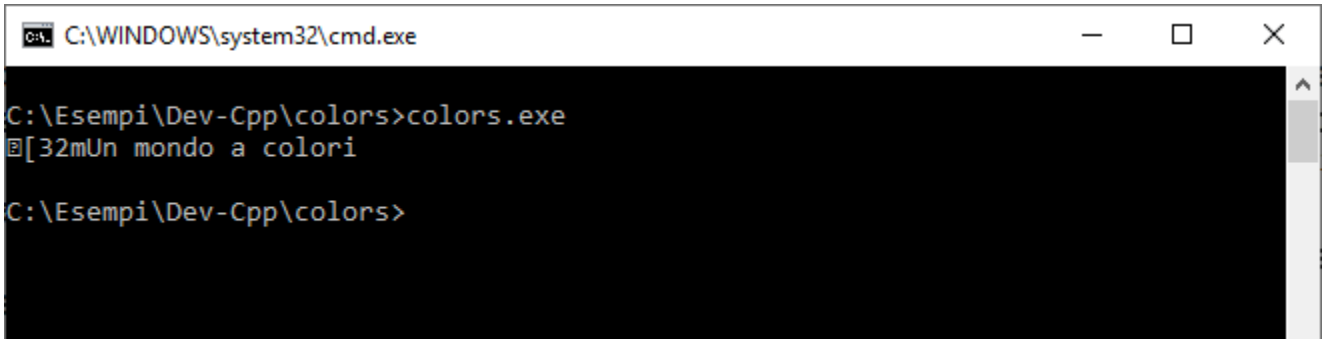

Figura - L'esecuzione con l'invio dei codici di escape con la cmd.exe.

Come si può osservare, in output troviamo in maniera grezza parte della sequenza inviata senza però nessuna modifica di colore. Al contrario, se lanciamo lo stesso eseguibile in un contesto Unix like, ad esempio nella finestra Bash del sottosistema Windows per Linux (configurabile sotto Windows 10) otterremo la corretta interpretazione del comando grafico, così come mostrato in figura:

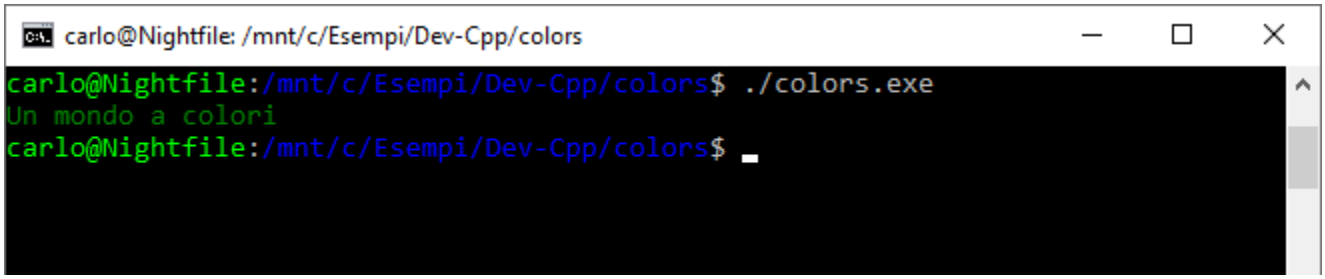

Figura - L'esecuzione con l'invio dei codici di escape nel sottosistema Windows per Linux.

La cosa interessante da notare è che una volta impartito un certo comando di impostazione di un dato colore, tale setting rimane invariato finché non decidiamo di inviare il codice per un colore diverso oppure per resettare il sistema al bianco e nero inviando il valore zero. Di seguito, vi riporto del codice che dovrebbe chiarire il meccanismo in questione:

```
#include <stdio.h>

int main ()
{
    printf("\x1b[32mUn mondo a colori\n");
    printf("... tutto verde\n");
    printf("\x1b[0m");
    printf("ma fino a un certo punto.\n");
}
```

In output otterremo, come mostrato in Figura, che le prime due righe risulteranno in verde mentre la riga di testo finale, "`ma fino a un certo punto.`", verrà mostrata in bianco su nero a causa del comando di reset:

```
"\x1b[0m"
```

dove riconosciamo ancora la sequenza `\x1b` che invia in esadecimale in carattere `Esc`, poi la parentesi quadra aperta e quindi il valore zero per il reset seguito dalla lettera `m` che segnala, come già detto, l'invio di un comando di tipo grafico.

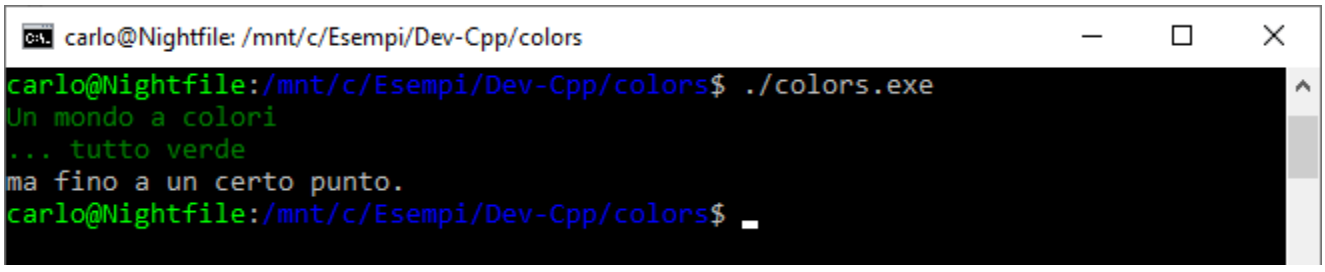

Figura - L'esecuzione con l'invio dei codici per il reset delle impostazioni di colore predefinite.

Vediamo ora di scoprire l'elenco dei colori che possiamo gestire. Ebbene, i colori di base solo sostanzialmente solo otto ed i loro codici vanno da 30 a 37. Tuttavia, gli stessi otto colori possono essere utilizzati per cambiare il colore di sfondo della console con dei codici che vanno da 40 a 47. Di seguito vi propongo uno schema riassuntivo.

COLORE	CODICE TESTO	CODICE SFONDO
Nero	30	40
Rosso	31	41
Verde	32	42
Giallo	33	43
Blu	34	44

Magenta	35	45
Azzurro	36	46
Bianco	37	47

Ovviamente, volendo cambiare contemporaneamente il colore del testo e quello dello sfondo dobbiamo inviare due specifici e differenti comandi. Ad esempio, volendo impostare il testo nero su sfondo bianco dovremo inviare, rispettivamente, i valori 30 e 47. Di seguito un semplice esempio:

```
#include <stdio.h>

int main ()
{
    printf("\x1b[30m");
    printf("\x1b[47m");
    printf("Un mondo al contrario.\n");
    printf("\x1b[0m");
}
```

Incidentalmente, vi faccio notare come sarebbe possibile inserire i due comandi in un'unica stringa di testo accodando le due sequenze di escape scrivendo:

```
printf("\x1b[30m\x1b[47m");
```

Appurato che gli ambienti di derivazione Unix gestiscono senza problemi le sequenze di escape vediamo ora cosa ci riserva il mondo Windows. Purtroppo la possibilità di usare in maniera quasi naturale tali sequenze è limitata alle sole versioni di Windows 10.

NOTA. Vedi anche `https://docs.microsoft.com/en-us/windows/console/console-virtual-terminal-sequences`.

Vediamo allora un programmino minimale che imposta l'ambiente al fine di utilizzare le sequenze in questione:

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

#ifndef ENABLE_VIRTUAL_TERMINAL_PROCESSING
#define ENABLE_VIRTUAL_TERMINAL_PROCESSING  0x0004
#endif

int main()
{
    //Catturo l'handle del dispositivo standard di output
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);

    //uso tale handle per ottenere l'attuale modalità della console
    DWORD dwMode = GetConsoleMode(hOut, &dwMode);

    //Abilito le sequenze di escape mettendo in OR alla modalità corrente
    //del buffer dello schermo della console il valore ENABLE_VIRTUAL_TERMINAL_PROCESSING
   dwMode = dwMode | ENABLE_VIRTUAL_TERMINAL_PROCESSING;

   //Imposto la nuova modalità con la funzione  SetConsoleMode
   SetConsoleMode(hOut, dwMode);

   printf("\x1b[32mUn mondo a colori\n");
   printf("... tutto verde\n");
   printf("\x1b[0m");
   printf("ma fino a un certo punto.\n");

   return 0;
}
```

Come detto, l'esecuzione di tale codice su sistemi antecedenti ad un aggiornato Windows 10 fallirà miseramente.
Proviamo allora ad analizzare il codice in questione partendo dalla necessità di definire la costante `ENABLE_VIRTUAL_TERMINAL_PROCESSING`. Usiamo quindi le cosiddette include guard `#ifndef` per evitare la eventuale redefinizione della costante in questione che viene richiesta dal sistema per poter gestire in maniera corretta le sequenze di escape.
Per il resto, il commento nel codice dovrebbe essere sufficiente a comprendere il meccanismo di funzionamento del sistema in questione. Quello che facciamo è innanzitutto prelevare il riferimento (noto come handle) del dispositivo di output, ovvero della console, con l'istruzione:

```
HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
```

Usiamo allora questo handle per poter leggere l'attuale impostazione della console:

```
DWORD dwMode = GetConsoleMode(hOut, &dwMode);
```

Incidentalmente, segnalo che un `dword`, abbreviazione di "double word," è un tipo di dati specifico di Windows. Tale tipo è definito nel file `windows.h` che abbiamo incluso in testa al nostro codice. Un `dword` è un intero senza segno a 32 bit e può quindi contenere un valore che va da 0 a 4.294.967.295.

Successivamente, aggiungiamo, tramite l'operatore `OR` la nuova impostazione al contesto corrente, usando la funzione `SetConsoleMode`:

```
dwMode = dwMode | ENABLE_VIRTUAL_TERMINAL_PROCESSING;
SetConsoleMode(hOut, dwMode);
```

A questo punto, il gioco è fatto e possiamo inviare le sequenze di escape di nostro interesse verso il dispositivo console.

La gestione degli errori

Ovviamente, per rendere il codice più robusto dovremmo preoccuparci di controllare con apposito trapping di errore se le operazioni richieste vanno a buon fine. A tal fine, vi segnalo come si possa usare la funzione `GetLastError()` per catturare l'eventuale errore generato dalle varie chiamate viste in precedenza. Per essere più chiaro, vi propongo la riscrittura del codice precedente con l'inserimento di vari check per testare se le operazioni richieste vanno o meno a buon fine.

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

#ifndef ENABLE_VIRTUAL_TERMINAL_PROCESSING
#define ENABLE_VIRTUAL_TERMINAL_PROCESSING  0x0004
#endif

int main()
{
    //Catturo l'handle del dispositivo standard di output
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hOut == INVALID_HANDLE_VALUE)
    {
        printf("Funzione GetStdHandle - Errore: %d\n", GetLastError());
    }

    //uso tale handle per ottenere l'attuale modalità della console
    DWORD dwMode = 0;
    if (!GetConsoleMode(hOut, &dwMode))
    {
```

```
        printf("Funzione GetConsoleMode - Errore: %d\n", GetLastError());
    }

    //Abilito le sequenze di escape mettendo in OR alla modalità corrente
    //del buffer dello schermo della console il valore ENABLE_VIRTUAL_TERMINAL_PROCESSING
   dwMode = dwMode | ENABLE_VIRTUAL_TERMINAL_PROCESSING;

   //Imposto la nuova modalità con la funzione  SetConsoleMode
   if (!SetConsoleMode(hOut, dwMode))
    {
        printf("Funzione SetConsoleMode - Errore: %d\n", GetLastError());
    }


   printf("\x1b[32mUn mondo a colori\n");
   printf("... tutto verde\n");
   printf("\x1b[0m");
   printf("ma fino a un certo punto.\n");

   return 0;
}
```

Come si vede, effettuiamo il controllo in relazione alla chiamate delle varie funzioni e, nel caso di effettivo errore, stampiamo con `GetLastError` lo specifico numero di errore. Ovviamente, per verificare sul campo tali situazioni dobbiamo porci in un contesto che generi effettivamente un errore. In figura vi mostro quello che può succedere in ambiente Windows 7 che, come detto, non consente l'uso delle sequenze di escape.

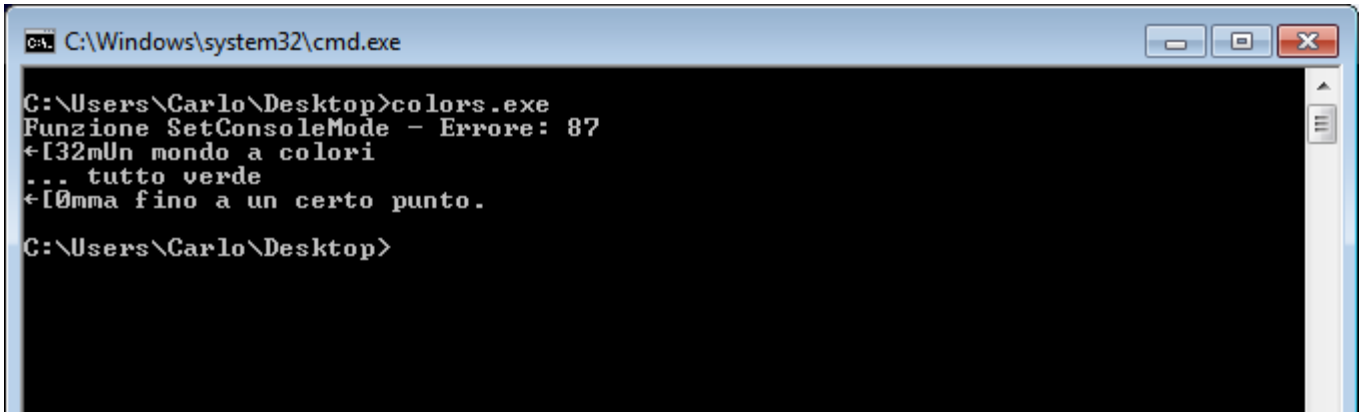

Figura - La generazione e il relativo trapping di un errore con le sequenze di escape sotto Windows 7.

Come si vede dalla figura in questione, l'errore generato è il numero 87. Spulciando la documentazione Microsoft, ad esempio all'URL `https://docs.microsoft.com/en-us/windows/win32/debug/system-error-codes--0-499-`, si scopre che tale numero corrisponde all'errore "`The parameter is incorrect.`". In ogni caso, può essere utile sapere che è possibile rendere la gestione degli errori ancora più espressiva utilizzando la funzione `FormatMessage` per farsi restituire la specifica stringa descrittiva dello specifico errore.

Su Windows usiamo le API

Una valida alternativa su Windows per controllare i colori della console può essere quella di sfruttare una funzione dell'API di Windows nota come `SetConsoleTextAttribute`. Di seguito vi mostro la sua struttura e modalità di utilizzo.
Il suo prototipo è:

```
BOOL SetConsoleTextAttribute(
    HANDLE hConsoleOutput,    // handle del buffer dello schermo della console
    WORD wAttributes     // colori per il testo e lo sfondo
);
```

Per rendere immediatamente comprensibile il suo funzionamento vi mostro subito un semplice esempio:

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

int main() {

    HANDLE output = GetStdHandle(STD_OUTPUT_HANDLE);

    printf("Questo e' il colore di default\n");
    SetConsoleTextAttribute(output, FOREGROUND_RED|FOREGROUND_INTENSITY);
    printf("Il colore rosso e' proprio bello\n");
    SetConsoleTextAttribute(output, FOREGROUND_BLUE|FOREGROUND_INTENSITY);
    printf("ma anche il blu non e' male\n");
    return 0;
}
```

La prima cosa che possiamo notare è l'include relativo al file `windows.h`. Subito dopo, con la riga:

```
HANDLE output = GetStdHandle(STD_OUTPUT_HANDLE);
```

otteniamo un handle per il buffer dello schermo. Usiamo tale handle come primo argomento della funzione `SetConsoleTextAttribute` mentre come secondo argomento impostiamo, in due momenti diversi, due differenti colori con le costanti `FOREGROUND_RED` e `FOREGROUND_BLUE`. Banalmente si tratta dei colori rosso e blu. Vi faccio notare come si possa accodare, con l'operatore `OR`, rappresentato dal simbolo "`|`" (noto come **pipe**), un ulteriore attributo rappresentato dalla costante `FOREGROUND_INTENSITY`. Tale costante serve per far sì che il colore appena selezionato sia visualizzato con una maggiore luminosità.
Ovviamente, i colori disponibili sono diversi. Di seguito un estratto riassuntivo:

COLORE	TESTO	SFONDO
Blu	`FOREGROUND_BLUE`	`BACKGROUND_BLUE`
Verde	`FOREGROUND_GREEN`	`BACKGROUND_GREEN`
Rosso	`FOREGROUND_RED`	`BACKGROUND_RED`
Intensificazione del colore	`FOREGROUND_INTENSITY`	`BACKGROUND_INTENSITY`

NOTA. Per ulteriori informazioni sui buffer della console `https://docs.microsoft.com/en-us/windows/console/console-screen-buffers`.

Un altro elemento interessante da analizzare è dato dal fatto che le precedenti costanti di colore possono essere combinate, con l'operatore OR, non solo con la costante per esaltare la luminosità del colore stesso ma anche con altre costanti di colore per ottenere varie e differenti combinazioni. Ad esempio, la combinazione:

```
BACKGROUND_BLUE | BACKGROUND_GREEN | BACKGROUND_RED
```

produrrà uno sfondo bianco che potrà essere esaltato in intensità accodando la costante `BACKGROUND_INTENSITY` e scrivendo quindi un comando del tipo:

```
SetConsoleTextAttribute(output, BACKGROUND_BLUE | BACKGROUND_GREEN | BACKGROUND_RED | BACKGROUND_INTENSITY);
```