Glordim


# Termcap et Terminfo

21 janvier 2019

# Table des matières



% TERMCAP ET TERMINFO % Glordim % 03 mai 2018

## 1. Introduction

Bienvenue à tous !
Dans ce tutoriel, nous étudierons deux bibliothèques assez méconnues mais pourtant très répendues : **Termcap** et **Terminfo**.
Sans entrer dans les détails disons qu'elles vous permettront, entre autre, d'étendre les capacités « graphiques » de votre terminal.

Et qui dit terminal dit forcement **Linux** ou **Unix**.
Inutile donc d'aller plus loin si vous n'êtes pas sur un de ces systèmes.

## 2. Le terminal

Bien !
Pour commencer, nous allons parler du terminal et de la manière dont il fonctionne. Nous n'entrerons pas dans les détails car ce n'est pas le but de ce tutoriel, mais certains points mériteront tout de même que l'on s'y attarde...

## 2.1. Les limites du terminal

On imagine souvent qu'un terminal n'est bon qu'à une chose : afficher des lignes de texte blanc sur fond noir et, pour ne rien vous cacher, on est assez proche de la vérité...

Au risque d'en décevoir plus d'un, oui un terminal n'est bon qu'à afficher du texte. Dites donc adieux à vos perspectives artistiques, vous ne pourrez ni afficher d'image ni contrôler précisément chaque pixel qui compose votre terminal.

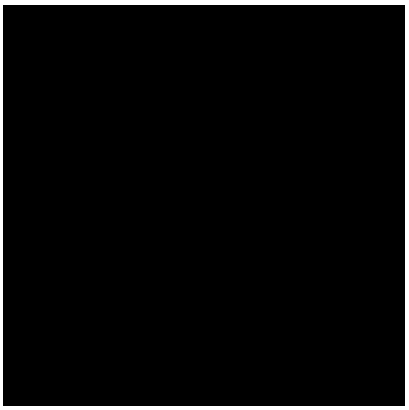

FIGURE 2. – Un terminal

En revanche, votre terminal est capable d'afficher des couleurs, mais aussi de placer le curseur à un endroit précis pour redessiner par dessus d'autres caractères !

Pour mieux visualiser la chose, imaginez que votre terminal est en fait un écran de plus ou moins 80 pixels de large sur 40 de haut et que chacun de ces « gros pixels » est capable d'afficher un caractère composé de deux couleurs (*foreground* et *background*).

Pour ceux qui auraient encore des doutes quant aux possibilités qu'offrent Termcap et Terminfo, voici un petit exemple d'application utilisant uniquement le changement de couleur et le déplacement du curseur.

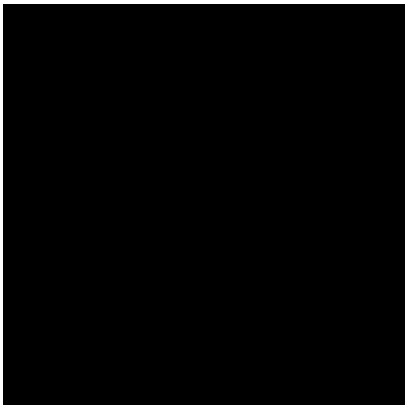

FIGURE 2. – Le jeu *Tetris* dans un terminal

Sur le même principe on pourrait très bien imaginer un *Snake*, un *Morpion*, un *Puissance 4*, un *Casse Brique*, un *Tron* et bien d'autres encore !

Vous pouvez aussi utiliser Termcap et Terminfo dans un cadre beaucoup plus sérieux en utilisant la colorisation pour afficher vos erreurs en rouge ou en déplaçant le curseur pour redessiner l'avancement d'une barre de progression. C'est vous qui voyez.

## 2.2. Les *termcaps*

Attardons-nous maintenant sur la manière dont un programme communique avec un terminal. Utilise t-il une librairie particulière ? Envoie t-il des signaux ou passe t-il par des sockets ?

Rien de tout cela, c'est en réalité bien plus simple.
Si on y réfléchit bien, nos programmes communiquent déjà avec leur terminal pour y afficher du texte. Et pour y parvenir, ils utilisent tout simplement la sortie standard (*stdout*) qui est, si aucune redirection n'est effectuée, directement envoyée à leur terminal pour y être affichée.

? Donc si je comprends bien, on contrôle un terminal en lui envoyant du texte ?

Oui, c'est presque ça. Il faut bien différencier le texte qui sera affiché tel quel du texte qui sera interprété et exécuté comme une commande. Dès l'instant où votre terminal reçoit du texte, il l'analyse et adopte le comportement adéquat.

Afin de mieux différencier les chaînes de caractères contenant une commande de celles contenant du texte, il a été décidé de les nommer ***termcap*** pour « *terminal capability* ». Retenez bien ce terme car il sera employé tout au long de ce tutoriel !

i Pour éviter toute confusion lors de la lecture, j'écrirai toujours *termcaps* en italique pour faire référence aux commandes et Termcap avec une majuscule pour la bibliothèque.

? D'accord, mais si je m'amuse à afficher le contenu d'un fichier texte, n'y a t-il pas un risque pour qu'un morceau du texte soit interprété comme un *termcap* par mégarde ?

Malheureusement, oui, étant donné que les *termcaps* ne sont que de simples chaînes de caractères, il y a effectivement un risque pour qu'ils soient exécutés totalement par hasard...

Bon après il ne faut pas non plus exagérer la chose.
Les *termcaps* sont assez repoussant et il n'y a quasiment aucune chance d'en croiser dans un texte destiné à un humain normalement constitué.

Pour ceux qui auraient encore des doutes, voici à quoi ressemble un *termcap* : `\033[1;31m`

! Attention ! Le risque est minime pour un fichier texte mais ce n'est pas le cas pour un fichier binaire ou crypté ! Il m'est déjà arrivé d'exécuter un *termcap* en faisant un `cat` sur

un fichier binaire... Si vous vous retrouvez un jour sans curseur ou avec un texte de couleur différente utilisez la commande `reset` pour restaurer votre terminal dans son état initial.

## 3. Termcap et Terminfo

Commençons par un peu d'histoire...
En 1978, un certain Bill Joy eut la bonne idée d'écrire une bibliothèque permettant de charger et d'utiliser les *termcaps* de manière portable. Il faut comprendre qu'à cette époque chaque terminal définissait et implémentait ses propres *termcaps* et il y avait parfois de fortes disparités d'un terminal à l'autre !

C'est donc sans surprise que la première bibliothèque dédiée à l'utilisation des *termcaps* prit le nom de Termcap !

Et comme par enchantement, de plus en plus d'Unix se mirent à adopter Termcap et au fil du temps tous les terminaux finirent par s'aligner et adoptèrent une sorte de norme commune.

Mais l'histoire ne s'arrête pas là...
En 1981, Mark Horton créa Terminfo, une autre bibliothèque très similaire à Termcap. En comparaison, Terminfo était plus performante et pris peu à peu le dessus grâce à son modèle de stockage plus complet. Au fil des années Termcap fut peu à peu délaissé au profit de Terminfo et l'engouement fut tel qu'aujourd'hui **Termcap est marqué comme obsolète sur certains systèmes** !

J'insiste bien sur le « sur certains systèmes » !
Historiquement, les systèmes basés sur *System V* utilisent Terminfo alors que les systèmes dérivés de *BSD* préfèrent encore Termcap mais finalement beaucoup ont fait le choix de supporter les deux.

---

Bien, revenons à nos clémentines !
Pour couvrir le plus de systèmes possibles et pour ne laisser personne sur la paille, ce tutoriel traitera de Termcap et de Terminfo en simultané. Les deux bibliothèques ont un fonctionnement très similaire et les séparer en deux tutos distincts n'aurait pas eu de sens. Pour preuve voici un petit tableau comparatif de leurs fonctions.

| Termcap | Terminfo |
|---|---|
| tgetent | setupterm |
| tgetstr | tigetstr |
| tgetnum | tigetnum |
| tgetflag | tigetflag |
| tparm | tiparm |
| tgoto | tgoto |

| tputs | tputs |
|---|---|

C'est tout ? On peut vraiment faire un *Tetris* avec ces quelques fonctions ?

Oui, vous comprendrez les raisons de cette légèreté lorsque je vous expliquerai la manière dont Termcap et Terminfo fonctionnent.

En attendant passons à l'installation des prérequis.
Vous possédez surement déjà une partie des paquets requis car Termcap et Terminfo sont aujourd'hui directement inclus dans la bibliothèque NCurses ! En revanche vous devrez vous munir des paquets de développement pour compiler vos programmes.

Sur Debian : `apt-get install lib32ncurses5-dev libncurses5-dev ncurses-doc`
(je vous laisse trouver l'équivalent sur votre distribution)

Pour information, il existe une solution très largement utilisée aujourd'hui qui a remplacé Termcap et Terminfo pour la création d'interface, il s'agit des bibliothèques Curses et NCurses.

- Curses est une bibliothèque développée par Ken Arnold qui utilise en arrière plan Termcap pour la création d'interface graphique dans un terminal. Son développement s'arrêta avec la version 4.4BSD.
- NCurses est une version libre et améliorée de Curses, initialement crée sous le nom de PCurses en 1982, elle changea de nom pour célébrer sa première release en 1993. NCurses est encore maintenue aujourd'hui et la dernière version majeure (6.0 à l'heure ou j'écris ces lignes) est parue en août 2015.

Là où NCurses tire son épingle du jeu, c'est qu'elle ré-implémente en interne Termcap et Terminfo de manière transparente tout en ajoutant énormément de fonctionnalités pour la gestion d'interface graphique. Mais ce n'est pas le sujet de ce tutoriel, la bibliothèque NCurses est tellement vaste qu'il lui faudrait un tutoriel entièrement dédié !

# 4. Initialisation

Entrons maintenant dans le vif du sujet !
Pour commencer nous allons voir comment initialiser Termcap et Terminfo.

## 4.1. Termcap

Comme beaucoup d'autres bibliothèques, Termcap et Terminfo nécessitent d'être initialisées au démarrage de votre programme. Dans le cas de Termcap, la fonction qui nous intéresse est `tgetent`, jetons un œil à son prototype (`man tgetent`).

```c
int tgetent(char *bp, const char *name);
```

D'après la doc, cette fonction prend en premier paramètre l'adresse d'un *buffer* dans lequel stocker les informations utiles au fonctionnement interne de Termcap et en second paramètre le nom / le type de votre terminal.

? Pourquoi Termcap à besoin du nom de mon terminal ?

Ce qu'il faut savoir, c'est que **Termcap et Terminfo agissent en fait comme des bases de données** contenant les *termcaps* propres à chaque terminal. Si vous souhaitez déplacer le curseur avec Xterm, le *termcap* à exécuter ne sera pas forcement le même qu'avec Gnome-term !

Termcap et Terminfo auront donc besoin de connaitre le type de terminal sur lequel vous exécutez votre programme pour aller charger les *termcaps* correspondant. Un peu comme si chaque terminal parlait sa propre langue.

Pour le plus curieux, sachez que ces fameuses bases de données sont stockées ici `/etc/termcap` et là `/usr/share/lib/terminfo`. En y regardant de plus près, vous verrez qu'elles sont organisées par type de terminal.

Bon, revenons en à notre initialisation de `tgetent`.
Fort heureusement, il existe une variable d'environnement qui contient le nom du terminal utilisé par votre programme !

`env | grep TERM`

Chez moi il s'agit de Xterm.
Récupérons cette info et stockons là quelque part pour la passer en argument à `tgetent`.

```c
char *term_type = getenv("TERM");
```

Bien !
Il ne nous manque plus qu'un *buffer* dans lequel stocker les jeux d'instructions propres à notre type de terminal ou du moins c'est ce que nous aurions dû faire il y a bien longtemps... D'après la doc, dans les implémentations modernes de Termcap, il n'est plus nécessaire de passer ce pointeur car le *buffer* est directement géré en interne par NCurses.

```c
##include <stdlib.h>

##include <curses.h>
#include <term.h>

int main(int argc, char **argv)
{
```

```
 8      int ret;
 9      char *term_type = getenv("TERM");
10
11      ret = tgetent(NULL, term_type);
12
13      /* On évite les warnings pour variables non utilisées. */
14      (void)argc;
15      (void)argv;
16
17      return ret;
18  }
```

Nous y sommes presque, il ne reste plus qu'à contrôler la valeur de retour de `tgetent` pour savoir si tout s'est bien passé et, dans le cas contraire, quitter proprement en avertissant l'utilisateur.

`tgetent` possède trois valeurs de retour (dont deux fatales) :

- `-1`, la base de données n'est pas accessible.
- `0`, la base de données est accessible mais elle ne contient pas d'info pour ce type de terminal ou trop peu pour fonctionner convenablement.
- `1`, la base de données est accessible et toutes les infos pour ce terminal ont été chargées en mémoire.

---

Revoici le code de l'initialisation **avec une gestion des erreurs** :

```
 1  ##include <stdlib.h>
 2  #include <stdio.h>
 3
 4  ##include <curses.h>
 5  #include <term.h>
 6
 7  int init_term()
 8  {
 9      int ret;
10      char *term_type = getenv("TERM");
11
12      if (term_type == NULL)
13      {
14          fprintf(stderr, "TERM must be set (see 'env').\n");
15          return -1;
16      }
17
18      ret = tgetent(NULL, term_type);
19
20      if (ret == -1)
21      {
```

```
22          fprintf(stderr,
                "Could not access to the termcap database..\n");
23          return -1;
24      }
25      else if (ret == 0)
26      {
27          fprintf(stderr,
                "Terminal type '%s' is not defined in termcap database (or have too
                term_type);
28          return -1;
29      }
30
31      return 0;
32  }
33
34  int main(int argc, char **argv)
35  {
36      int ret = init_term();
37
38      /* On évite les warnings pour variables non utilisées. */
39      (void)argc;
40      (void)argv;
41
42      if (ret == 0)
43      {
44          /* Le déroulement de notre programme se fera ici. */
45      }
46
47      return ret;
48  }
```

## 4.2. Terminfo

Passons maintenant à l'initialisation de Terminfo !
Rassurez-vous, elle est bien plus simple que celle de Termcap ! Voyons son prototype (man setupterm).

```
1  int setupterm(char *term, int fildes, int *errret);
```

Tout comme `tgetent`, `setupterm` demande aussi en premier paramètre le type de terminal utilisé. En second, il demande sur quel descripteur de fichier (*file descriptor*) afficher sa sortie. Puis en troisième et dernier argument un `int*` pour y inscrire un éventuel code d'erreur.

Mais la bonne nouvelle dans tous ça, c'est que tous ces paramètres sont complètements facultatifs (ou presque) !

— Si on passe un pointeur nul en premier argument, `setupterm` ira elle même chercher le contenu de la variable d'environnement « TERM ».
— On se contentera de lui passer la sortie standard (*stdout*) en tant que deuxième argurment.
— Si on passe un pointeur nul en troisième argument, il affichera lui même les erreurs qu'il aura rencontrées.

En soi, on s'en sortira très bien avec cette simple ligne :

```c
##include <stdlib.h>
#include <stdio.h>

##include <curses.h>
#include <term.h>
##include <unistd.h>

int main(int argc, char **argv)
{
    int ret = setupterm(NULL, STDOUT_FILENO, NULL);

    /* On évite les warnings pour variables non utilisées. */
    (void)argc;
    (void)argv;

    if (ret == 0)
    {
        /* Le déroulement de notre programme se fera ici. */
    }

    return ret;
}
```

---

N'oubliez pas de lier la bibliothèque NCurses quand vous compilerez votre programme : `-lncurses`

## 5. Tget et Tputs

Maintenant qu'on a vu comment initialiser Termcap et Terminfo, nous allons pouvoir nous attaquer à la partie la plus intéressante du tuto : **tget** et **tputs**.

---

### 5.0.1. tgetnum / tigetnum

```c
/* Termcap */
int tgetnum(char *id);

/* Terminfo */
int tigetnum(char *capname);
```

`tgetnum` permet de récupérer des informations numériques en rapport avec votre terminal. Comme par exemple le nombre de lignes et de colonnes.

Pour l'utiliser rien de plus simple, il suffit de lui passer en argument ce que l'on souhaite récupérer :

```c
/* Termcap */
int column_count = tgetnum("co");
int line_count = tgetnum("li");

/* Terminfo */
int column_count = tigetnum("cols");
int line_count = tigetnum("lines");
```

---

### 5.0.2. tgetflag / tigetflag

```c
/* Termcap */
int tgetflag(char *id);

/* Terminfo */
int tigetflag(char *capname);
```

`tgetflag` fonctionne de la même manière que `tgetnum` à la différence prêt qu'il renvoie un booléen au lieu d'une valeur. Cette fonction est utilisée pour vérifier les capacités d'un terminal, savoir s'il est capable de faire tel ou tel action.

```c
/* Termcap */
if (tgetflag("os") != 0)
{

}
```

```
6
7  /* Terminfo */
8  if (tigetflag("os") != 0)
9  {
10
11 }
```

---

### 5.0.3. tgetstr / tigetstr

```
1  /* Termcap */
2  char *tgetstr(char *id, char **area);
3
4  /* Terminfo */
5  char *tigetstr(char *capname);
```

`tgetstr` est de loin la fonction la plus appréciable des **tget**.
C'est elle qui permet de récupérer les fameux *termcaps* sous la forme d'une séquence d'échappement ! On peut par exemple récupérer le *termcap* « cl » (pour ***cl****ean*) qui permet de nettoyer (vider) un terminal.

`tgetstr` prend en deuxième paramètre l'adresse du *buffer* que l'on a utilisé pour `tgetent`, à savoir `NULL`.

```
1  /* Termcap */
2  char *cl_cap = tgetstr("cl", NULL);
3
4  /* Terminfo */
5  char *cl_cap = tigetstr("clear");
```

---

### 5.0.4. tputs

```
1  int tputs(const char *str, int affcnt, int (*putc)(int));
```

`tputs` est la fonction qui marche de pair avec `tgetstr`, c'est elle qui va se charger d'exécuter le *termcap* que l'on vient de récupérer.

Le premier argument de `tputs` correspond bien évidement au *termcap* à exécuter, le second au nombre de lignes affectées et le troisième à un pointeur de fonction pour afficher le contenu du *termcap* (`putchar` fera amplement l'affaire).

```c
/* Termcap */
char *cl_cap = tgetstr("cl", NULL);
tputs (cl_cap, 1, putchar);

/* Terminfo */
char *cl_cap = tigetstr("clear");
tputs (cl_cap, 1, putchar);
```

Pour rappel, `putchar` se comporte comme `printf` et stocke tout son contenu dans un *buffer*. Par conséquent, la véritable « impression » ne se fait qu'à la réception d'un '\n' ou manuellement en appelant `fflush(stdout)`. Si vous recherchez une solution immédiate, créez votre propre fonction en utilisant un `write` (appel système).

---

Vous l'aurez remarqué, Termcap et Terminfo, en plus d'avoir des fonctions similaires, ont aussi une base données quasiment identique. Le nom de leurs *termcaps* change mais ils proposent au final les mêmes fonctionnalités.

Voici un petit tableau pour mettre en évidence cette ressemblance.
J'y ai listé les *termcaps* qui à mes yeux sont les plus utiles, mais pas de panique, nous allons tous les voir un par un.

| Termcap | Terminfo | Description |
|---|---|---|
| co | cols | nombre de colonnes affichées à l'écran |
| li | lines | nombre de lignes affichées à l'écran |
| AF | setaf | définit la couleur du texte |
| AB | setab | définit la couleur de fond |
| md | bold | affiche le texte en « gras » |
| us | smul | affiche le texte en « souligné » |
| mb | blink | affiche le texte en « clignotant » |
| cm | cup | déplace le curseur aux coordonnées souhaitées |
| cl | clear | efface le texte affiché à l'écran |
| me | sgr0 | annule tous les changements opérés |
| os | os | *over strike*, précise si le terminal efface ou non le contenu lors d'une réécriture par dessus du texte (par exemple à la suite d'un *backspace*) |

# 6. De la couleur

Maintenant que vous avez compris le principe des *tget* / *tputs*, il est temps de mettre tout ça en application et quoi de mieux pour cela que la couleur !

Commençons par récupérer le *termcap* correspondant : « AF ».
(« AF » correspond à la couleur du texte, utilisez « AB » pour la couleur du fond)

```c
/* Termcap */
char *af_cap = tgetstr("AF", NULL);

/* Terminfo */
char *af_cap = tigetstr("setaf");
```

Mais cette fois, un simple `tputs` ne suffira pas. Nous allons d'abord devoir paramétrer notre *termcap* pour qu'il sache quelle couleur afficher ! Heureusement, il existe une fonction parfaitement adaptée pour ça : `tparm` !

```c
/* Termcap */
char *tparm(char *str, ...);

/* Terminfo */
char *tiparm(const char *str, ...);
```

`tparm` prend en premier argument le *termcap* à paramétrer et en second argument une `va_list`.

> Pour ceux qui l'ignoreraient, une `va_list` est une sorte de liste d'arguments à taille dynamique, on peut lui passer 1, 2, 3 ou N arguments. Pour en savoir plus sur ce procédé, n'hésitez pas à aller jeter un œil au tuto C .

Mais avant d'utiliser `tparm`, jetons un œil au *termcap* que nous à retourner `tgetstr`.

```c
void print_escape_sequence(char *str)
{
        unsigned i = 0;

        while (str[i] != '\0')
        {
                if (str[i] == '\x1b')
                        printf("ESC");
                else
                        putchar(str[i]);
```

```
11
12                  ++i;
13          }
14
15          putchar('\n');
16  }
```

Si on tente d'afficher la séquence d'échappement correspondant au *termcap* AF sous Xterm, on obtient ceci :

`ESC[3%p1%dm`

En s'attardant un peu sur ce résultat, on s'aperçoit que notre séquence est composée de deux éléments étrangement familier que l'on retrouve également lorsqu'on utilise `printf` (qui lui aussi utilise une `va_list`) : `%p` et `%d`.

Et oui ! Quand on utilise `printf`, on passe d'abord le texte puis un nombre indéfini d'arguments pour remplacer les éléments dynamiques de ce texte (`%d`, `%f`, `%p`, etc...) par leur valeur. Hé bien avec `tparm` c'est exactement la même chose ! (à la différence près qu'on ne passe pas la valeur pour le premier `%p`)

Dans notre cas le *termcap* « AF » ne prend donc qu'un argument (`%d`) : la couleur !

```c
/* N'hésitez pas à utiliser les defines des couleurs inclues dans
    curses.h ! */
##include <curses.h>

/* Termcap */
char* color_cap = tgetstr("AF", NULL);
tputs(tparm(color_cap, COLOR_GREEN), 1, putchar);

printf("Cool ! Maintenant j'ecris en vert !\n");

/* Terminfo */
char* color_cap = tigetstr("setaf");
tputs(tiparm(color_cap, COLOR_RED), 1, putchar);

printf("Cool ! Maintenant j'ecris en rouge !\n");
```

Attention, le changement de couleur est permanent.
Quand votre programme se termine **votre terminal reste dans l'état dans lequel vous l'avez laissé** !
Pensez donc à réinitialiser ses attributs avec la commande « me ».

```c
/* Termcap */
char *reset_cmd = tgetstr("me", NULL);
tputs(reset_cmd, 1, putchar);

/* Terminfo */
char *reset_cmd = tigetstr("sgr0");
tputs(reset_cmd, 1, putchar);
```

---

En dehors de la couleur, il y a d'autres effets sympas à expérimenter sur du texte. Essayez les commandes « **mb** », « **us** » et « **md** » qui vous permettront respectivement de rendre un texte clignotant, souligné ou écrit en gras.

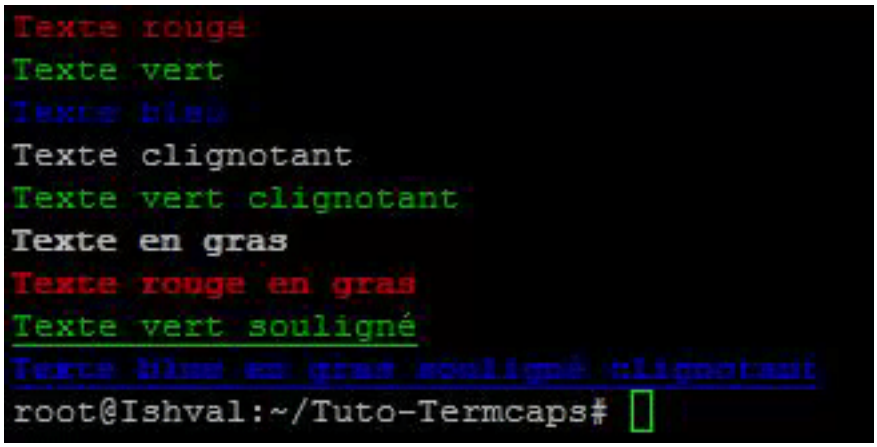


FIGURE 6. – Désolé pour la médiocre qualité du GIF...

Voici un code d'exemple pour Termcap, je vous laisse trouver les *capnames* correspondants pour Terminfo.

Contenu masqué n°1

## 7. Déplacer le curseur

Attaquons-nous maintenant au déplacement du curseur !
Comme d'habitude, commençons par récupérer le *termcap* assosié : « **cm** ».

```c
/* Termcap */
char *cm_cap = tgetstr("cm", NULL);

```

```c
/* Terminfo */
char *cm_cap = tigetstr("cup");
```

Comme pour la couleur, il va falloir paramétrer la commande pour y inclure les coordonnées auxquelles vous souhaitez déplacer votre curseur mais cette fois il s'agit d'un *termcap* un peu particulier, on n'utilisera pas `tparm` mais `tgoto` !

En soi, ça ne change pas grand chose hormis que `tgoto` prend deux paramètres bien définis au lieu d'une `va_list`.

```c
char *tgoto(const char *cap, int col, int row);
```

Et comme d'habitude, on le couplera avec un `tputs` :

```c
/* Déplace le curseur en 5,5 */
tputs(tgoto(cm_cap, 5, 5), 1, putchar);
```

> *i* Le fait de pouvoir placer le curseur à un endroit précis devrait vous ouvrir de nombreuses portes. On pourrait par exemple l'utiliser pour effacer une forme dans un *Tetris* et la redessiner plus bas.

## 8. Aller plus loin

Vous l'aurez compris, il existe de nombreuses autres *termcaps* à expérimenter.
Dans ce tutoriel nous avons vu ensemble ceux qui, à mes yeux, étaient les plus intéressants et je vous encourage à en découvrir d'autres même si je pense que ceux présentés ici devraient déjà répondre à la plupart de vos besoins !

Pour la documentation, je vous recommande de jeter un œil à cette page ou à celle-ci .
En soi ce ne sont que des `man 5 terminfo` sur une page web mais ça reste quand même bien pratique !

## 9. Le défi des agrumes : Tetris

La théorie c'est bien, mais la pratique c'est mieux !
Je vous propose donc de vous faire la main avec un Tetris, depuis le temps que j'en parle ce n'est plus vraiment une surprise...

Pour le réaliser vous aurez besoin de Termcap / Terminfo pour :

— Nettoyer l'écran.

— Placer votre curseur aux bons endroits.
— Gérer la couleur pour dessiner vos formes.

Vous devrez aussi configurer votre terminal correctement pour qu'il n'affiche pas bêtement tout ce que vous tapez sur votre clavier. Il faudra aussi penser à gérer le temps et les entrées utilisateurs (`read` + `select`) ! Mais pas de panique, pour cet exercice, je vous fournirai ces quelques morceaux de code qui n'ont aucun rapport avec Termcap ou Terminfo et qui seraient peut-être un poil trop avancé pour certains néophytes.

#### 9.0.0.1. Configuration du terminal avec Termios :

Contenu masqué n°2

#### 9.0.0.2. Récupération des entrées clavier avec read et select :

Contenu masqué n°3

**Bon courage !**

---

La correction est disponible sur Github .
Elle n'est pas parfaite mais rien ne vous empêche de l'améliorer en ajoutant par exemple :

— Le score.
— L'augmentation de la vitesse.
— L'affichage de la forme suivante.
— Un *game over*.

## 10. Conclusion

Voilà, c'est la fin de ce petit tutoriel sur Termcap et Terminfo.
J'espère qu'il vous aura plu et qu'il vous permettra de dompter votre terminal !

Pour ceux qui n'en auraient pas eu assez ou qui voudraient aller encore plus loin, je leur conseille de se tourner vers la bibliothèque NCurses qui est, à mon sens, la suite logique de ce tuto.

Bonne continuation à tous.
Et n'hésitez pas à nous faire partager vos créations sur le forum !

# Contenu masqué

## Contenu masqué n°1

```c
void testTextColorAndEffect()
{
        char *af_cmd = tgetstr("AF", NULL);
        char *reset_cmd = tgetstr("me", NULL);
        char *blink_cmd = tgetstr("mb", NULL);
        char *bold_cmd = tgetstr("md", NULL);
        char *underline_cmd = tgetstr("us", NULL);

        tputs(tparm(af_cmd, COLOR_RED), 1, putchar);
        printf("Texte rouge\n");

        tputs(tparm(af_cmd, COLOR_GREEN), 1, putchar);
        printf("Texte vert\n");

        tputs(tparm(af_cmd, COLOR_BLUE), 1, putchar);
        printf("Texte bleu\n");

        tputs(reset_cmd, 1, putchar);
        tputs(blink_cmd, 1, putchar);
        printf("Texte clignotant\n");

        tputs(tparm(af_cmd, COLOR_GREEN), 1, putchar);
        printf("Texte vert clignotant\n");

        tputs(reset_cmd, 1, putchar);
        tputs(bold_cmd, 1, putchar);
        printf("Texte en gras\n");

        tputs(tparm(af_cmd, COLOR_RED), 1, putchar);
        printf("Texte rouge en gras\n");

        tputs(reset_cmd, 1, putchar);
        tputs(tparm(af_cmd, COLOR_GREEN), 1, putchar);
        tputs(underline_cmd, 1, putchar);
        printf("Texte vert souligné\n");

        tputs(tparm(af_cmd, COLOR_BLUE), 1, putchar);
        tputs(bold_cmd, 1, putchar);
        tputs(blink_cmd, 1, putchar);
        tputs(underline_cmd, 1, putchar);
        printf("Texte blue en gras souligné clignotant\n");
        tputs(reset_cmd, 1, putchar);
}
```

*Contenu masqué*

Retourner au texte.

## Contenu masqué n°2

```c
#include "termcap_initializer.h"
#include "tetris.h"

#include <termios.h>

int main(int argc, char **argv)
{
        struct termios s_termios;
        struct termios s_termios_backup;
        int ret = 0;

        (void)argc;
        (void)argv;

        if (init_termcap() == 0) /* Fonction que vous aurez créé
            avec un tgetent dedans. */
        {
                if (tcgetattr(0, &s_termios) == -1)
                        return (-1);

                if (tcgetattr(0, &s_termios_backup) == -1)
                        return (-1);

                s_termios.c_lflag &= ~(ICANON); /* Met le terminal
                    en mode non canonique. La fonction read recevra
                    les entrées clavier en direct sans attendre
                    qu'on appuie sur Enter */
                s_termios.c_lflag &= ~(ECHO); /* Les touches tapées
                    au clavier ne s'affficheront plus dans le
                    terminal */

                if (tcsetattr(0, 0, &s_termios) == -1)
                        return (-1);

                ret = tetris();

                if (tcsetattr(0, 0, &s_termios_backup) == -1)
                        return (-1);
        }

        return ret;
}
```

Retourner au texte.

## Contenu masqué n°3

```c
int i = 0;
char read_buffer[64] = {0};
struct timeval timeout;

fd_set read_fd_set;

FD_ZERO(&read_fd_set);
FD_SET(STDIN_FILENO, &read_fd_set);

timeout.tv_sec = 0;
timeout.tv_usec = 5000;

i = select(STDIN_FILENO + 1, &read_fd_set, NULL, NULL, &timeout);

if (i == -1)
{

}
else if (i == 0)
{

}
else
{
        if (FD_ISSET(i, &read_fd_set) != 0)
        {
                read(STDIN_FILENO, read_buffer,
                    sizeof(read_buffer));

                if (strncmp(read_buffer, tc_cmd.kl,
                    strlen(tc_cmd.kl)) == 0 || read_buffer[0] ==
                    'a') /* q for azerty ayout */
                {
                        /* Ici on déplace la forme à gauche */
                }
                else if (strncmp(read_buffer, tc_cmd.kr,
                    strlen(tc_cmd.kr)) == 0 || read_buffer[0] ==
                    'd')
                {
                        /* Ici on déplace la forme à droite */
                }
                else if (strncmp(read_buffer, tc_cmd.kd,
                    strlen(tc_cmd.kd)) == 0 || read_buffer[0] ==
                    's')
                {
                        /* Ici on déplace la forme vers le bas */
                }
```

*Contenu masqué*

```
41                  else if (strncmp(read_buffer, tc_cmd.ku,
                        strlen(tc_cmd.ku)) == 0 || read_buffer[0] ==
                        'w') /* z for azerty layout */
42                  {
43                          /* Ici on fait tourner la forme */
44                  }
45          }
46  }
```

Retourner au texte.