# Saisie interactive de textes en C

June 3, 2024

## 1 Objectif

Saisir **des textes**, i.e. plusieurs à la suite, dont des textes **vides**, en **C**, avec **scanf**, éventuellement sur les 2 [ou 3] systèmes (Unix, [MacOs], Windows).

Remarques :

- Pourquoi `scanf` ? Parce que c'est la primitive de lecture la plus connue/utilisée (elle est même conseillée aux débutants par les IA).
- Il s'agit de "textes", des espaces peuvent être présents, des textes vides aussi.
- Pour délimiter la fin d'une saisie, on considérera que la touche `Entrée` clos la saisie (rem. : en générale, elle ajoute aussi un passage à la ligne à la saisie, mais ce `"\n"` n'est pas voulu), ou que la saisie est close si le flux d'entrée est fini (`EOF`).
- Lire des *nombres* ou des *caractères* ou des textes *non vides* ou *un seul* texte, c'est une autre histoire
- Dans les pages de manuel Unix/Linux récentes (changement autours des années 2020-23), l'utilisation de `scanf` est déconseillée. La fonction `scanf` est considérée comme d'usage difficile (et source de bug ?), lui sont préférées les fonctions `fgets` et `getline` (avec sscanf, à la suite [!])

## 2 Limites des solutions naïves

La solution naIve, avec `scanf("%s")` :

[17]:
```
%%writefile progNaif.c
#include <stdio.h>

int main() {
    char ch[81];

  printf("Chaine ?\n");
  scanf("%s",ch);
  printf(">> Chaine = >>%s<<\n\n", ch);

  printf("Chaine ?\n");
  scanf("%s",ch);
  printf(">> Chaine = >>%s<<\n", ch);

return 0;}
```

```
Overwriting progNaif.c
```

### 2.1 `scanf("%s")` s'arrête aux séparateurs blancs (espaces, tabulation, etc.)

Donnons deux textes avec des blancs.

[18]:
```
!gcc progNaif.c -o progNaif.e
!printf "Bon jour \n la terre\n" | ./progNaif.e
```

```
Chaine ?
>> Chaine = >>Bon<<

Chaine ?
>> Chaine = >>jour<<
```

Il manque la moitié des entrées.

### 2.2 `scanf("%s")` n'accèpte pas les réponses vides

Pour une première entrée "vide" :

[15]:
```
!gcc progNaif.c -o progNaif.e
!printf "\nBonjour\n" | ./progNaif.e
```

```
Chaine ?
>> Chaine = >>Bonjour<<

Chaine ?
>> Chaine = >>Bonjour<<
```

La première entrée/ligne était vide, la saisie l'a omise (c'est peut-être une sage solution [dans certains cas]) et la saisie a pris le seconde ligne comme première réponse, mais si la première entrée/réponse voulait être **vide**, c'est une erreur, la saisie fournit en première réponse la seconde entrée, et en seconde réponse conserve cette entrée .

Remarques :

- s'il y avait eu une troisème entrée, elle aurait été prise comme seconde réponse.

Pour une seconde entrée "vide" :

[9]:
```
!gcc progNaif.c -o progNaif.e
!printf "Bonjour\n\n" | ./progNaif.e
```

```
Chaine ?
>> Chaine = >>Bonjour<<

Chaine ?
>> Chaine = >>Bonjour<<
```

La seconde ligne/réponse était vide, à nouveau la réponse "vide" est mal interprétée, c'est comme une erreur, et comme une erreur la variable `ch` est laissée inchangée (pourquoi pas, mais ici, cela signifie que `ch` reste avec la première saisie ! alors qu'elle devrait être vide).

## 3 Analyse du problème

Différents problèmes principaux empêchent d'avoir une solution "simple" avec `scanf("%s")` :

- Les blancs terminent la saisie des `scanf("%s")`
- Les saisies vides sont considérées comme des échecs et laissent inchangées les paramètres
- Selon les OS, il faut passer un ou deux caractères (et pas le même en premier) pour éviter les délimiteurs de fin de saisie apparaissant sous forme de caractères de fin de ligne

Il faut donc utiliser autre chose que `scanf("%s")`, passer les fin de lignes, et prévoir les possibles saisie vides.

## 4 Résolution

[20]:
```
%%writefile prog.c
#include <stdio.h>
int main() {
  char ch[81];

  printf("Chaine ?\n");
  ch[0]=0;                    // en cas de saisie vide, le scanf va échouer,␣
 ↪mais "ch" aura la bonne valeur !
  scanf("%80[^\n\r]",ch);     // lecture jusqu'à la fin de la saisie (\n, \r ou␣
 ↪EOF)
  scanf("\r");                // lecture des délimiteurs de fin de saisie, si␣
 ↪présents
  scanf("\n");                //   '' idem '' (dans l'ordre naturel \r\n de␣
 ↪Windows)
  printf(">> Chaine = >>%s<<\n\n", ch);

  printf("Chaine ?\n");
  ch[0]=0;
  scanf("%80[^\n\r]",ch);
  scanf("\r");
  scanf("\n");
  printf(">> Chaine = >>%s<<\n\n", ch);
return 0;}
```

```
Overwriting prog.c
```

## 5 Conclusion

Tests :

[21]:
```
!gcc prog.c -o prog.e
!printf "Bon jour \n la terre\n" | ./prog.e
```

```
Chaine ?
>> Chaine = >>Bon jour <<

Chaine ?
>> Chaine = >>la terre<<
```

[26]:
```
!gcc prog.c -o prog.e
!printf "Bonjour la terre\n\n" | ./prog.e
```

```
Chaine ?
>> Chaine = >>Bonjour la terre<<

Chaine ?
>> Chaine = >><<
```

[23]:
```
!gcc prog.c -o prog.e
!printf "\nBonjour la terre\n" | ./prog.e
```

```
Chaine ?
>> Chaine = >><<

Chaine ?
>> Chaine = >>Bonjour la terre<<
```

[25]:
```
!gcc prog.c -o prog.e
!printf "Bon jour \r\n la terre\r\n" | ./prog.e
```

```
Chaine ?
>> Chaine = >>Bon jour <<

Chaine ?
>> Chaine = >>la terre<<
```

Ca fonctionne.

D'autres solutions sont possibles avec `fgets`, `readline` et des expressions régulières (ou même avec d'autres traitements utilisant `scanf`).

L'avantage de la solution présente, c'est qu'elle passe facilement en assembleur.

```
@ avec l'adresse de ch dans R1
ldr r0, LD_FormatLigne          @ "%80[^\n\r]"
mov R2, #0
strb R2, [R1]                   @ ch[0]=0
bl scanf
ldr r0, LD_FormatCR             @ "\r"
```

```
bl scanf
ldr r0, LD_FormatLF             @ "\n"
bl scanf
```