Entrées solides en langage C

par Emmanuel Delahaye (Espace personnel d'Emmanuel Delahaye)

Date de publication : 27 janvier 2008

Dernière mise à jour : 25 avril 2009

Cet article a pour but de mettre en évidence le comportement des fonctions de saisie, et d'élaborer des fonctions d'entrées solides à partir des fonctions d'entrées unitaires.

Votre avis et vos suggestions sur cet article nous intéressent !
Alors après votre lecture, n'hésitez pas :

I - Introduction

Le langage C dispose de fonctions d'entrées standards dont l'usage est parfois surprenant, que ce soit par le comportement parfois inattendu de certaines fonctions, ou plus gravement par le risque de débordement lorsqu'une quantité de données trop importante est entrée.

Cet article a pour but de mettre en évidence le comportement des fonctions de saisie, et d'élaborer des fonctions d'entrées solides à partir des fonctions d'entrées unitaires.

II - Les fonctions d'entrées standards

Ce sujet est détaillé dans l'article **Saisie de données par un opérateur (stdin)**. Il en ressort que la fonction de saisie unitaire est fgetc(), et qu'elle couvre tous les cas, puisqu'elle se contente d'extraire un byte du flux d'entrée.

Son fonctionnement détaillé est expliqué dans l'article **Comment fonctionne fgetc(stdin) alias getchar()**

III - Mise en évidence du fonctionnement de fgetc()

Voici un petit programme qui permet de vérifier le fonctionnement de fgetc().

Dans le cas où stdin est connecté au clavier, la fin de lecture (EOF) est provoquée au clavier par la frappe d'une commande spéciale qui dépend du système :

- MS-DOS, Windows : <Ctrl-Z><enter> (en debut de ligne)
- Unixoides : <Ctrl-D>

```
/* 01/main.c */

#include <stdio.h>

int main (void)
{
   int c;

   while ((c = fgetc (stdin)) != EOF)
   {
      fputc (c, stdout);
   }
   return 0;
}
```

Tous les essais suivants sont réalisés sous Windows XP avec l'IDE Code::Blocks (Mingw)

La frappe de 'abcdef<enter><Ctrl-Z><enter>' provoque cette sortie console :

```
abcdef
abcdef
^Z

Press ENTER to continue.
```

Afin de mieux 'voir ce qui se passe', on ajoute quelques éléments de visualisation (debug)

```
/* 02/main.c */

#include <stdio.h>
#include <ctype.h>

int main (void)
{
   int c;

   while ((c = getchar ()) != EOF)
   {
      if (isprint (c))
      {
         printf ("'%c'   (0x%02X)\n", c, (unsigned) c);
      }
      else
      {
         switch (c)
         {
         case '\n':
            printf ("'\\n' (0x%02X)\n", (unsigned) c);
            break;
         case '\t':
            printf ("'\\t' (0x%02X)\n", (unsigned) c);
            break;
         default:
```

```
            printf ("??  (0x%02X)\n", (unsigned) c);
         }
      }
   }
   return 0;
}
```

On obtient maintenant

```
abcdef
'a'  (0x61)
'b'  (0x62)
'c'  (0x63)
'd'  (0x64)
'e'  (0x65)
'f'  (0x66)
'\n' (0x0A)
^Z

Press ENTER to continue.
```

On remarque que la frappe de <enter> provoque la fin de la suspension, et que l'intégralité des caractères frappés est extraite et affichée, y compris le '\n' qui marque la fin de la ligne.

On peut aussi constater que si on corrige sa frappe avec la touche <backspace>, la correction est gérée en interne, et qu'aucun caractère '\b' n'apparaît dans la liste des caractères entrés.

Enfin, on peut aussi constater que si on entre plusieurs lignes, celle-ci seront traitées complètement à chaque fois que l'on entre le caractère de fin de ligne (frappe de <enter>). Par exemple :

```
abc
'a'  (0x61)
'b'  (0x62)
'c'  (0x63)
'\n' (0x0A)
defg
'd'  (0x64)
'e'  (0x65)
'f'  (0x66)
'g'  (0x67)
'\n' (0x0A)
^Z

Press ENTER to continue.
```

Remarque importante. Le nombre de caractères saisi en une ligne peut être très important, et tant qu'il y a des caractères à lire, la boucle continue à les extraire.

Sous MS-DOS/Windows, il y a cependant une limite de 127 caractères (au-delà, le système émet un bip et la saisie est bloquée).

Mais cette limite est bien supérieure (voire indéterminée) sur une machine unixoide ou Windows NT. Il est donc prudent de ne faire aucune hypothèse sur une éventuelle limitation.

Voici un petit code qui permet de compter les caractères entrés à chaque ligne:

```
/* 03/main.c */

#include <stdio.h>

int main (void)
{
   int c;
```

```
   unsigned count = 0;

   while ((c = fgetc (stdin)) != EOF)
   {
      fputc (c, stdout);
      count ++;

      if (c == '\n')
      {
         printf ("%lu byte%s read\n", count, count > 1 ? "s" : "");
         count = 0;
      }
   }

   return 0;
}
```

Ce qui donne, par exemple :

```
1 byte read
abcd
abcd
5 bytes read
efghijkl
efghijkl
9 bytes read
^Z

Press ENTER to continue.
```

Sous XP, j'ai pu saisir une ligne de plus de 800 caractères sans problèmes...

IV - Réalisation d'une fonction de lecture de lignes()

IV-A - Stocker les données reçues

S'agissant de caractères, on va tout naturellement utiliser un tableau de char. Une question de conception se pose alors immédiatement : quelle taille donner au tableau ? Il n'y a pas de réponse universelle à cette question, à part "une taille infinie", ce qui n'a évidemment aucun sens. Dans un premier temps, on se contentera donc de la réponse laconique "une taille raisonnable". Evidemment, on prendra les précautions indispensables pour ne pas déborder du tableau.

Exemple : saisie d'un nom (32 caractères au maximum)

```c
/* 10/main.c */
#include <stdio.h>
#include <assert.h>

int main (void)
{
   int c;

   /* bytes counter */
   unsigned long count = 0;

   /* index position */
   unsigned i = 0;

   /* array of char to store the line */
   char line[32 + 1];

   /* secure the array: put a sentinel */
   line[sizeof line - 1] = 0;

   while ((c = fgetc (stdin)) != EOF)
   {
      /* check the limit */
      if (i < sizeof line - 1)
      {
         /* store the data */
         line[i] = c;

         /* check the sentinel */
         assert (line[sizeof line - 1] == 0);

         /* next index position */
         i++;
      }
      else
      {
         puts ("full");
      }

      /* bytes counter */
      count++;

      if (c == '\n')
      {
         /* terminate the C-string */
         line[i] = 0;

         /* check the sentinel */
         assert (line[sizeof line - 1] == 0);

         /* print the line */
         printf ("'%s'\n", line);

         /* print the last line index */
         printf ("%u byte%s stored\n", i, i > 1 ? "s" : "");
```

```
            i = 0;

            /* print the bytes counter */
            printf ("%lu byte%s read\n", count, count > 1 ? "s" : "");
            count = 0;
         }
      }

      return 0;
}
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
'aaaaaaaaaaaaaaaaaaaaaaaaaaaa
'
30 bytes stored
30 bytes read
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
full
full
full
full
full
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
32 bytes stored
37 bytes read
^Z

Press ENTER to continue.
```

Lors de la première saisie, l'ensemble des caractères lus, y compris le '\n' a été stocké et affiché.

Lors de la deuxième saisie, il y a eu 'saturation', et le programme indique que les cinq derniers caractères entrés ont été lus, mais non stockés. Ils sont perdus. Le '\n' est absent de la chaine 'ligne'.

Nous pouvons donc affirmer que ce code est sûr, car :

- il interdit le débordement du tableau
- il lit tous les caractères saisis
- il constitue une chaine valide

IV-B - Créer une fonction de saisie

IV-B-1 - Définition générale

Soit à réaliser une fonction en C standard permettant la saisie d'une ligne de texte. L'utilisateur fournit un espace mémoire et la taille de celui-ci. La fonction assure la saisie des caractères, la lecture dans la mesure du possible, le stockage sous forme d'une chaine valide (sans '\n' et avec 0 final) et la purge des caractères non lus. Elle retourne une valeur indiquant le bon fonctionnement ou non.

Dans le mesure du possible, tous les comportements seront définis, quelque soient les contraintes extérieures et les défauts systèmes.

La contrainte d'implémentation "en C standard" implique l'usage de fgetc() (ou une fonction dérivée de celle-ci), ce qui définit par avance une partie importante du comportement

IV-B-2 - Définition de l'interface

La fonction est nommée 'get_line'.

```
get_line()
```

Elle retourne un int valant 0 en cas de succès, ou une valeur supérieure à 0 en cas d'echec.

```
int get_line()
```

Elle admet 2 paramètres : l'adresse du premier élément du tableau de destination et la taille de celui-ci. Les éléments du tableau étant de type char, le paramètre recevant l'adresse est de type char*. Le paramètre recevant la taille est de type size_t.

```
int get_line(char *s, size_t n)
```

IV-B-3 - Définition du comportement

Une grande partie du comportement découle de celui de la fonction standard fgetc() qui sera utilisée pour l'implémentation. Lorsque la fonction est appelée, l'exécution du programme est suspendue en attente de l'entrée d'un caractère '\n' en provenance de stdin (Généralement, le clavier).

L'opérateur peut alors saisir des caractères. Il peut les modifier avec les commandes d'éditions standard du système (BACKSPACE, par exemple). Lorsqu'il veut terminer la saisie, il appuye sur la touche qui signifie 'fin de saisie' sur son système (ENTER, par exemple).

Les caractères sont alors lus jusqu'à premier '\n' rencontré (il correspond à l'appui de la touche 'fin de saisie'). Ils sont stockés dans la zone mémoire fournie par l'utilisateur à partir de l'indice 0 de s et ce, dans la limite de n - 1 caractères. Dans tous les cas si m caractères ont été placés dans le tableau, un 0 est placé à l'indice m de s.

La valeur retournée par défaut est 0. Une valeur supérieure à 0 est retournée en cas d'erreur.

A ce stade de la définition, il est difficile de prévoir tous les cas d'erreurs. Ceux-ci seront détaillés lors de l'implémentation

- 0 : Pas d'erreur
- 1 : Paramètre erroné (pointeur NULL ou taille < 1)
- 2 : Un débordement potentiel a été detecté (la taille maximale a été atteinte et des caractères autres que '\n' ont donc été lus sans être stockés)
- 3 : Une fin de saisie brutale a été détectée (la saisie a été brutalement interrompue par l'opérateur, par exemple par l'appui de Ctrl-Z ou Ctrl-D selon le système)

IV-B-4 - Conception

L'algorithme suivant décrit le comportement demandé (les cas d'erreurs ne sont pas traités ici)

```
FAIRE
 lire un caractère
 SI la place est suffisante
  le placer dans le tableau
 FIN SI
TANT QUE le caractère de fin de ligne, n'a pas été detecté
Placer un 0 dans le tableau après le dernier caractère stocké.
```

Il serait possible de décrire un algorithme plus détaillé, mais étant donné sa simplicité, l'implémentation directe devrait suffire.

IV-B-5 - Codage et tests unitaires

L'implémentation est faite selon la méthode XP, à savoir la programmation par contrat. Un test unitaire est associé à chaque étape du développement. Ces étapes sont détaillées ici :

IV-B-5-a - Interface, contrôle des paramètres

```
/* 11/main.c */
#include <stdio.h>

/* device under test */

/* interface */
int get_line(char *s, size_t n);

/* implementation */
int get_line(char *const s, size_t const n)
{
   int err = 0;

   if (s != NULL && n > 0)
   {
      /* to be continued ... */
   }
   else
   {
      err = 1;
   }

   return err;
}

/* unit test */

#define NELEM(a) (sizeof(a)/sizeof *(a))

int main (void)
{
   struct test
   {
      int tnum;

      /* parameters */
      char *s;
      size_t n;

      /* return */
      int err;
   };

   static char s[32];
   static const struct test a[] =
      {
         {
            1, NULL, 0, 1
         },
         {
            2, s, 0, 1
         },
         {
            3, s, 1, 0
         },
         {
            4, s, sizeof s, 0
         },
      };
   size_t i;
   int terr = 0;
```

```
   for (i = 0; i < NELEM (a) && !terr; i++)
   {
      struct test const *const p = a + i;

      int err = get_line (p->s, p->n);

      if (err != p->err)
      {
         printf ("ERR at test %d\n", p->tnum);
         terr = 1;
      }
   }

   if (!terr)
   {
      puts ("\nP A S S E D\n");
   }

   return 0;
}
```

IV-B-5-b - Boucle de saisie, stockage

On ajoute la boucle de saisie et le stockage avec contrôle de limites.

```
/* 12/main.c */
#include <stdio.h>

/* device under test */

/* interface */
int get_line(char *s, size_t n);

/* implementation */
int get_line(char *const s, size_t const n)
{
   int err = 0;

   if (s != NULL && n > 0)
   {
      int c;
      size_t w = 0;

      while ((c = fgetc (stdin)) != '\n' && c != EOF)
      {
         if (w < n - 1)
         {
            s[w] = c;
            w++;
         }
         else
         {
            err = 2;
         }
      }
      s[w] = 0;
   }
   else
   {
      err = 1;
   }

   return err;
}

/* unit test */
#include <string.h>
```

```c
#define NELEM(a) (sizeof(a)/sizeof *(a))

int main (void)
{
   struct test
   {
      int tnum;

      /* parameters */
      char *s;
      size_t n;

      /* return */
      int err;

      /* input string */
      char const *sin;

      /* output string */
      char const *sout;

   };

   static char s[8];
   static const struct test a[] =
      {
         /* wrong parameters */
         {
            1, NULL, 0, 1, NULL, NULL
         },
         {
            2, s, 0, 1, NULL, NULL
         },

         /* normal parameters */
         {
            10, s, sizeof s, 0, "", ""
         },
         {
            11, s, sizeof s, 0, "1", "1"
         },
         {
            12, s, sizeof s, 0, "1234567", "1234567"
         },
         {
            13, s, sizeof s, 2, "12345678", "1234567"
         },
         {
            14, s, sizeof s, 0, "abc", "abc"
         },
      };
   size_t i;
   int terr = 0;

   for (i = 0; i < NELEM (a) && !terr; i++)
   {
      struct test const *const p = a + i;

      int err;

      if (p->sin != NULL)
      {
         printf ("Test %d : entrer '%s' puis <ENTER>\n", p->tnum, p->sin);

         memset (s, '?', sizeof s);
         s[sizeof s - 1] = 0;
      }

      err = get_line (p->s, p->n);

      if (err != p->err)
      {
```

```
            printf ("ERR at test %d\n", p->tnum);
            terr = 1;
         }
         else
         {
            if (p->sin != NULL)
            {
               if (strcmp(s, p->sout) != 0)
               {
                  printf ("ERR at test %d: s='%s' sout='%s'\n", p->tnum, s, p->sout);
                  terr = 1;
               }
               else
               {
                  printf ("OK: '%s'\n", s);
                  printf ("err = %d\n", err);
               }
            }
         }
      }

      if (!terr)
      {
         puts ("\nP A S S E D\n");
      }

      return 0;
}
```

IV-B-5-c - Finalisation. Constantes pour les erreurs, traitement du <EOF>, compilation séparée

Fichier d'interface **get_line.h**

```
#ifndef H_GET_LINE
#define H_GET_LINE

/* get_line.h */
#include <stddef.h>

/* return codes */
enum
{
   GET_LINE_OK,
   GET_LINE_ERR_PARAM,
   GET_LINE_ERR_TOO_LONG,
   GET_LINE_ERR_EOF,
   GET_LINE_ERR_NB
};

/* interface */
int get_line(char *s, size_t n);

#endif /* guard */
```

Fichier d'implémentation **get_line.c**

```
/* get_line.c */

#include "get_line.h"
#include <stdio.h>

/* implementation */
int get_line(char *const s, size_t const n)
{
   int err = GET_LINE_OK;

   if (s != NULL && n > 0)
   {
```

```
      int c;
      size_t w = 0;

      while ((c = fgetc (stdin)) != '\n' && c != EOF)
      {
         if (w < n - 1)
         {
            s[w] = c;
            w++;
         }
         else
         {
            if (!err)
            {
               err = GET_LINE_ERR_TOO_LONG;
            }
         }
      }
      s[w] = 0;

      if (c == EOF)
      {
         err = GET_LINE_ERR_EOF;
      }
   }
   else
   {
      err = GET_LINE_ERR_PARAM;
   }

   return err;
}
```

Fichier de test unitaire **main.c**

```
/* 13/main.c */

/* unit test for getline() */

#include "get_line.h"
#include <stdio.h>
#include <string.h>

#define NELEM(a) (sizeof(a)/sizeof *(a))

int main (void)
{
   struct test
   {
      int tnum;

      /* parameters */
      char *s;
      size_t n;

      /* return */
      int err;

      /* input string */
      char const *sin;

      /* output string */
      char const *sout;

   };

   static char s[8];
   static const struct test a[] =
      {
         /* wrong parameters */
         {
```

```
            1, NULL, 0, GET_LINE_ERR_PARAM, NULL, NULL
        },
        {
            2, s, 0, GET_LINE_ERR_PARAM, NULL, NULL
        },

        /* normal parameters */
        {
            10, s, sizeof s, GET_LINE_OK, "", ""
        },
        {
            11, s, sizeof s, GET_LINE_OK, "1", "1"
        },
        {
            12, s, sizeof s, GET_LINE_OK, "1234567", "1234567"
        },
        {
            13, s, sizeof s, GET_LINE_ERR_TOO_LONG, "12345678", "1234567"
        },
        {
            14, s, sizeof s, GET_LINE_OK, "abc", "abc"
        },
        {
            15, s, sizeof s, GET_LINE_ERR_EOF, "<EOF>", ""
        },
    };
size_t i;
int terr = 0;

for (i = 0; i < NELEM (a) && !terr; i++)
{
    struct test const *const p = a + i;

    int err;

    if (p->sin != NULL)
    {
        printf ("Test %d : entrer '%s' puis &ENTER>\n", p->tnum, p->sin);

        memset (s, '?', sizeof s);
        s[sizeof s - 1] = 0;
    }

    /* device under test */
    err = get_line (p->s, p->n);

    if (err != p->err)
    {
        printf ("ERR on returned value at test %d\n", p->tnum);
        terr = 1;
    }
    else
    {
        if (p->sin != NULL)
        {
            if (strcmp(s, p->sout) != 0)
            {
                printf ("ERR at test %d: s='%s' sout='%s'\n", p->tnum, s, p->sout);
                terr = 1;
            }
            else
            {
                printf ("OK: '%s'\n", s);
                printf ("err = %d\n", err);
            }
        }
    }
}

if (!terr)
{
    puts ("\nP A S S E D\n");
```

```
    }

    return 0;
}
```

V - Conclusion

Nous disposons maintenant d'une fonction de base permettant la saisie d'une ligne de taille fixe de façon simple et fiable. Des améliorations sont possibles, comme la généralisation à tout fichier texte (simple), ou la saisie d'une ligne de longueur arbitraire (un peu plus complexe, et mettant en oeuvre l'allocation dynamique).

Je laisse au lecteur le soin de poursuivre l'expérience. Des solutions plus ou moins complexes sont proposées **ici** (Module IO)