

Un système d'options

Par pylaterreur



www.openclassrooms.com

*Licence Creative Commons 2 2.0
Dernière mise à jour le 25/12/2009*

Sommaire


Sommaire	2
Un système d'options	3
Les options simples à un caractère	3
D'autres types d'options ?	4
Une gestion d'erreurs basique	5
Une "white-list"	5
Fonction de hachage	6
Fonction(s) de gestion des listes chaînées	6
Partager	11



Un système d'options

Par  pylaterreur

Mise à jour : 25/12/2009

Difficulté : Intermédiaire 



Bonjour à toi, ami lecteur 😊.

Vous connaissez et utilisez les options. Et oui, quand vous tapez "rm -rf foo.txt", vous indiquez 2 options : 'r' et 'f'. Ce tutoriel vous permettra de facilement stocker et récupérer ces options dans vos programmes en C.

Sommaire du tutoriel :



- [Les options simples à un caractère](#)
- [D'autres types d'options ?](#)

Les options simples à un caractère

Dans ce tutoriel, on va voir comment créer facilement un système d'options, comme il en existe pour de nombreux programmes. Par exemple, quand vous tapez :

Code : Bash

```
ls -la
```

Vous demandez en fait les options 'l' et 'a' du programme ls. A chaque option correspond donc un caractère.

Comment faire pour récupérer au cours de l'exécution ces différentes options ?

Une solution serait de parcourir à chaque fois le tableau des paramètres, mais ça peut être assez lent et pas forcément pratique si vous comptiez modifier le tableau des arguments.

Et si on stockait les options dans un tableau ? On sait qu'une option est représentée par un caractère, donc par un octet. Un octet peut avoir 2^8 (=256) valeurs différentes.

Une option est soit activée, soit désactivée. On peut donc dire qu'elle vaut soit 0 soit 1. Un char suffit amplement pour stocker ça.

On peut donc créer un tableau de 256 chars.

Une chaîne d'options commence par '-'. Si on rencontre un argument contenant "--" ou ne commençant pas par '-', on arrête le parcours.

Code : C

```
static char tab[1 << (sizeof(char) << 3)] = {0}; /* sizeof(char)
= 1; 1<<(1<<3) = 1*2^(1*2^3) = 2^8 = 256; */

void my_options(const int argc, const char **argv)
{
    int i;
    size_t j;

    for (i = 1; i < argc; ++i)
    {
```

```

        if (argv[i][0] != '-' || argv[i][1] == '-')
            return;
        for (j = 1; argv[i][j] != '\0'; ++j)
            tab[(size_t)(unsigned char)argv[i][j]] = 1;
    }
}

```

Ensuite, si vous avez besoin de savoir si l'option 'f' est activée, il suffit d'écrire `tab[(size_t)(unsigned char)'f']`, ce qui vous renverra soit 0, soit 1. C'est assez optimisé, il n'y a aucune boucle à faire (enfin, une seule, au début) 😊.

Le compilateur n'accepte pas (et il a raison) d'utiliser un char comme index. Le problème, c'est que nous on en a besoin. C'est pour cela que nous utilisons des [casts](#) pour forcer le compilateur.



Pourquoi deux casts ?

Mmm... `argv[i][j]` est un char signé. L'utilisateur peut rentrer toute sorte de char, pas forcément des positifs. Que se passe-t-il si vous convertissez un char négatif en (unsigned int) ?

Testez le code suivant, et essayez de comprendre par vous-même [en vous aidant du complément à 2](#) :

Code : C

```

#include <stdio.h>

int main(void)
{
    char c;

    c = -1;
    do
    {
        ++c;
        printf("c : %u\tc : %u\n", (unsigned int)(unsigned char)c,
            (unsigned int)c);
    }
    while (c + 1 != 0);
    return (0);
}

```

Pour ceux qui ne peuvent pas compiler, voici un extrait de ce qu'affiche le programme :

```

c : 0 c : 0
.....
c : 127 c : 127
c : 128 c : 4294967168
.....
c : 255 c : 4294967295

```



Le cast (unsigned char) avant le (unsigned int) nous sauve du buffer overflow. En effet, nous utilisons ce nombre comme index pour accéder à notre tableau de 256 chars, et 4294967295 ça ferait plus que déborder 🤖. En binaire, 127 s'écrit 01111111 (le bit de signe est à 0), et 128 s'écrit 10000000 (le bit de signe est à 1, ce qui explique que c'est la première ligne qui comporte une erreur).

D'autres types d'options ?

Pour les options longues (comme "--help"), vous pouvez garder le même principe, sauf qu'au lieu d'avoir un tableau de 256 chars, vous pouvez vous faire un tableau de listes chaînées, et accéder à la case qui vous intéresse en "hachant" la chaîne de

caractères qui sert de clef.

Je vous conseille de lire la page Wikipedia sur les [tables de hachage](#) 🤖.



Une table de hash (ou "table de hachage", ou encore "hash table") est une structure de données très pratique. Grâce à elle on peut récupérer des données en donnant un index autre qu'un nombre.

Si vous avez déjà fait du PHP, par exemple, vous avez pu utiliser les tableaux associatifs, où vous pouvez écrire `<?php $aFoo['clef'] = 'valeur';?>`. Une table de hash peut servir à ça, car elle permet de récupérer rapidement des données.

Et oui, on a utilisé une mini table de hachage dans la partie précédente. On n'utilisait pas une string comme clé, mais un char. Le principe reste néanmoins le même.

Pour les options à paramètres, pourquoi ne pas utiliser un tableau de 256 char* ?

Chaque option pointerait alors sur le argv suivant. Attention, `argv[argc]` vaut `NULL`.



Tout d'abord, de quoi avons-nous besoin pour faire un système d'options "complet" (courtes, longues, avec ou sans paramètres) ?

- Une gestion d'erreurs basique.
- Une liste des options que notre programme peut prendre, ainsi que leur type (courte ou longue, avec ou sans paramètres). La "white-list".
- Une fonction de hachage.
- Un système de gestion de listes chaînées (le nom est pompeux, mais le code sera simple 🤖).
- Une fonction stockant les différentes options rentrées à l'exécution.
- Une fonction récupérant les différentes options rentrées à l'exécution.

Une gestion d'erreurs basique

Là, ce sera vraiment basique : on jouera avec les retours de fonctions, et on créera une variable globale, qui contiendra un numéro d'erreur (un enum serait un plus, je vous laisse le faire de vous-même) :

Code : C

```
static size_t      erreur = 0;
```

Je vous laisse vous débrouiller avec cette variable, suivant vos besoins. La gestion d'erreurs que je vous propose est VRAIMENT basique. A vous de jouer 🤖.

Une "white-list"



Qu'est-ce qui définit une option ?

Je pense que vous serez d'accord avec moi, il y a tout d'abord son "nom" ("r" dans `rm -rf`, ou encore "help" dans `foo --help`), mais aussi son type ("help" est longue, tandis que "r" est courte, et le "f" du `tar -cvf file.tar file1` prend un paramètre, ici "file.tar"), ainsi que son état/paramètre (activée-désactivée/pointeur sur le paramètre-pointeur `NULL`).

Pour savoir si une option est "longue", il suffit de regarder le nombre de caractère qui la compose, pas besoin d'aller plus loin. En revanche, on ne peut pas deviner qu'une option prend un paramètre.

Je vous propose donc d'utiliser une structure que l'on définira comme ceci :

Code : C

```
typedef struct    s_option
{
    char          *name;                /* le nom de l'option */
    int           is_parameterized;     /* doit-elle prendre un
parametre ? */
    char          *param;              /* le pointeur sur le param
ou pointeur sur l'option en cours si option sans parametre */
    size_t        len;
};
```

```
}          t_option;
```

On créera ensuite un tableau de `t_option`, chaque case représentant une option. Une fois les options enregistrées, on ne les récupérera pas à partir de ce tableau, mais à partir d'un tableau de listes chaînées (qu'on appellera table de hachage). Pourquoi ne pas directement utiliser ce tableau ? Tout d'abord parce que je veux vous montrer comment utiliser une autre technique, et aussi parce que ça peut être utile si vous avez beaucoup d'options, avec des hash bien répartis, ce qui sera 256 fois plus rapide qu'un simple tableau pour récupérer une option 😊.

Code : C

```
/* exemple de white-list. on set le nom, le is_parameterized aux
valeurs qui nous interessent, et le param est, pour l'instant,
useless */
static t_option white_list[] = {
    {"r", 0, NULL},
    {"help", 1, NULL}
};
```

Fonction de hachage

On veut à partir d'une chaîne de caractères, accéder à une option. Je vous propose la fonction de hash suivante qui renvoie un `size_t` dans l'intervalle [0, 256[(sur 1 octet, en bref).

C'est VRAIMENT basique, "abc" donnera le même hash que pour "acb", "bac", "bca", "cab" ou "cba", ce qui est sans doute pas le top IRL, mais ça suffit pour ce tuto 😊.

Code : C

```
static size_t      hash(char *str)
{
    unsigned char c;

    for (c = 0; *str != '\0'; ++str)
        c += *str;
    return (c);
}
```

Fonction(s) de gestion des listes chaînées

Là, on n'a vu que la première moitié de la table de hachage, il reste encore les listes chaînées. Il y en aura 256.

Notre "data" sera une option, c'est-à-dire une variable de type `t_option`. Comme on est fainéant, on ne va pas copier les structures présentes dans la white-list, mais pointer dessus si on a besoin d'elles.

La structure qu'on utilisera pour la liste chaînée sera la suivante :

Code : C

```
typedef struct      s_lst
{
    struct s_option *data; /* pointeur sur notre data, declare dans
la white-list */
    struct s_lst *next; /* pointeur sur le maillon
suivant, pointe sur NULL si dernier de la liste */
} t_lst;
```

On aura la variable suivante (la hash-table) dans l'espace global :

Code : C

```
static t_lst          *hash_table[256];
```

Avant toute chose, vous n'oublierez pas de remettre tous les octets de la hash_table à 0 :

Code : C

```
/* `man memset` dit include de string.h, faites-le */
memset(hash_table, 0, sizeof(hash_table));
```

Pour stocker les options rentrées à l'exécution, j'ai voulu bien découper le travail. On aura :

- La fonction principale, prenant en paramètre le tableau de chaînes de caractère (la ligne de commandes) : `char *set_options(char *av[]);` .
- Une fonction insérant une option dans la table de hash : `static void insert_in_hash(t_option *white_list);` .
- Une fonction gérant un argument rempli d'options courtes : `static char *short_option(char **p);` .
- Une fonction gérant un argument contenant une option longue : `static char *long_option(char **p);` .
- Une fonction récupérant une option (un pointeur sur une option pour être exact) en fonction de son nom (elle hachera, accèdera à la case de la table de hachage, parcourra la liste en faisant des strcmp entre le paramètre reçu et le name de l'élément) : `static t_option *get_option(char *str);` .
- Une fonction "nettoyant" la table de hash, en fonction d'un critère (un pointeur sur fonction, prenant en paramètre un pointeur sur t_lst, ou pas de paramètre) : `static void clean_hash_table(int (*f)())`.

Code : C

```
/*
** Stocke les options activees dans la hash_table
** Retourne NULL s'il n'y a pas d'erreurs dans les options rentrees
*/
char *set_options(char *av[])
{
    size_t i;
    char **p;
    char *ret;

    /* on met tous les octets de hash_table a 0 */
    memset(hash_table, 0, sizeof(hash_table));
    /* on cree la table de hachage, qui contient TOUTES les options
    */
    for (i = 0; i < sizeof(white_list) / sizeof(*white_list); ++i)
    {
        white_list[i].len = strlen(white_list[i].name);
        insert_in_hash(white_list + i);
    }
    /* On parcourt les arguments a la recherche des options */
    for (p = av + 1; *p != NULL; ++p)
    {
        /*
        ** si on trouve un argument qui, soit :
        ** . ne commence pas par un '-'
        ** . est "--"
        */
        if (**p != '-' || !strcmp(*p, "--"))
            break;
        /* si ca commence par "--", on appelle long_option(), sinon
        short_option() */
        ret = ((p[0][1] == '-') ? long_option(&p) : short_option(&p));
        /*
        ** L'utilisateur a rentre :
```

```

** . une option invalide, ou
** . une option a parametre sans indiquer de parametre
*/
    if (ret != NULL)
return (ret);
}
/* On supprime de la table de hash les options non activees */
clean_hash_table(has_not_been_met);
return (NULL);
}

```

La fonction qui insère une option dans la table de hachage. Rien de bien mystérieux, c'est une pile (LIFO).

Code : C

```

/* Insere une option dans la table de hash :-o */
static void insert_in_hash(t_option *option)
{
    t_lst *lst;
    const size_t h = hash(option->name);

    lst = malloc(sizeof(*lst));
    /* et oui, c'est moche de faire un exit, mais c'est simple ^^ */
    if (lst == NULL)
        exit(EXIT_FAILURE);
    lst->next = hash_table[h];
    lst->data = option;
    hash_table[h] = lst;
}

```

Code : C

```

/* retourne NULL si erreur, sinon un pointeur sur t_option */
static t_option *get_option(char *str)
{
    const size_t h = hash(str);
    t_lst *p;

    /* on parcourt la liste a la case h de la hash_table */
    for (p = hash_table[h]; p != NULL; p = p->next)
        if (!strcmp(p->data->name, str))
            return (p->data);
    erreur = 1;
    return (NULL);
}

```

Code : C

```

/*
** Gere les options courtes
** char ***p pointe sur le pointeur pointant sur le parametre en
cours (qui est un pointeur sur char)
*/
static char *short_option(char ***p)
{
    /* **p pointe sur le premier caractere de l'option, le '-', c'est
pourquoi on ajoute strlen("-") (1) */
    char *str = **p + 1;
    t_option *option;
    char b[2] = {0};
    int i = 0;
}

```



```

/*
** on parcourt, tant qu'on a pas rencontre de '\0'
** si l'option est vide ("-"), on veut que ca genere une erreur,
** c'est pourquoi on utilise un do while et pas un while
*/
do
{
    *b = *str;
    /* on recupere l'option dont le nom est la string d'un
caractere (*str) */
    option = get_option(b);
    if (option == NULL)
return (str);
    if ((option->param = p[0][option->is_parameterized]) == NULL)
{
    erreur = 2;
    return (str);
}
    /* on retient s'il y a eu un parametre */
    if (option->is_parameterized)
i = 1;
    ++str;
}
while (*str != '\0');
/*
** si on a un parametre, l'argument suivant est un parametre,
** PAS une option, c'est pourquoi on avance *p de 1
*/
if (i)
    ++*p;
return (NULL);
}

```

Code : C

```

/* c'est plus simple que pour les options courtes : pas besoin de
boucle ;) */
static char *long_option(char ***p)
{
    /* **p pointe sur le premier caractere de l'option, le '-', c'est
pourquoi on ajoute strlen("--") (2) */
    char *str = **p + 2;
    t_option *option;

    option = get_option(str);
    if (option == NULL)
        return (str);
    if ((option->param = p[0][option->is_parameterized]) == NULL)
        return (str);
    if (option->is_parameterized)
        ++*p;
    return (NULL);
}

```

Vient enfin la clean_hash_table() :

Code : C

```

/* clean la hash-table en fonction de ce que renvoie f() */
static void clean_hash_table(int (*f)())
{
    size_t i;
    t_lst *old, *p, *next;

```

```

    for (i = 0; i < sizeof(hash_table) / sizeof(*hash_table); ++i)
    {
        old = NULL;
        for (p = hash_table[i]; p != NULL; p = next)
        {
            /* on sauvegarde le next, parce qu'on pourrait liberer p, et
            acceder a un membre d'une structure liberee, c'est pas top ^^ */
            next = p->next;
            /* on appelle f(), on lui envoie p, l'option en cours */
            if (f(p))
            {
                /* pour eviter que hash_table[i] pointe sur un t_lst libere
                */
                if (p == hash_table[i])
                hash_table[i] = next;
                free(p);
                /* si old != NULL, f(old) ==> 0, donc on ne l'a pas libere :
                c'est le dernier element vu dans la liste qu'on a pas libere */
                if (old != NULL)
                old->next = next;
            }
            else
                old = p;
        }
    }
}

```

En bonus, pour que ça compile 🤖, voici la fonction `has_not_been_met()`, utilisée dans `set_options()`.

Code : C

```

/* renvoie 1 si l'option pointee par p->data est activee, 0 sinon
*/
static int has_not_been_met(const t_lst *p)
{
    return ((p->data->param == NULL));
}

```

Et si vous voulez tester, rajoutez les 2 fonctions suivantes :

Code : C

```

static int all(void)
{
    return (1);
}

/* si ca refuse de compiler avec le __unused, enlevez-le, vous
aurez juste un warning */
int main(__unused int ac, char *av[])
{
    size_t i;
    char *ret;

    if ((ret = set_options(av)) != NULL)
    {
        fprintf(stderr, "Probleme avec l'option : \"%s\"\n", ret);
        return (EXIT_FAILURE);
    }
    for (i = 0; i < sizeof(white_list) / sizeof(*white_list); ++i)
    {
        printf("L'option \"%s\" est %sactivee\n", white_list[i].name,
        (white_list[i].param == NULL) ? "des" : "");
    }
}

```

```
    }  
    /* On a plus besoin de la table de hash : on free les lst alloues  
*/  
    clean_hash_table(all);  
    return (EXIT_SUCCESS);  
}
```

Ça paraît mastoc comme solution, ça l'est volontairement.

Vous savez utiliser une liste chaînée basique, un tableau, et donc une table de hachage (qui est un compromis mixant les 2 types de structures).

Et, accessoirement, vous avez un outil que vous pouvez "recycler" dans vos projets de programmation, un système de gestion d'options.

Vous pouvez améliorer ce système, de façon à ce que vous sachiez où commencent les vraies arguments, ceux qui se trouvent après les options.

Ça ne devrait pas être trop compliqué, dans `set_options()` vous savez quand vous quittez la boucle 😊.

[J'ai rassemblé pour vous les sources en un seul fichier .c.](#)

Et bien, voilà, le tuto est terminé. Vous avez de quoi gérer des options et faire mumuse avec des structures de données 😊.

La première partie était simple. La deuxième un peu moins, donc si vous n'avez pas tout compris, posez votre question dans les forums.

Si je me suis trompé quelque part, dites-le 😊.

Ah, j'allais oublier : faites un ``man 3 getopt`` sous Linux 😊.

Partager

