Les listes simplement chaînées en C

par Nicolas Joseph

Date de publication : 29 Juin 2005

Dernière mise à jour : 24 Août 2005

Les structures de données en C. Première partie : les listes simplement chaînées.

Historique

24 Août 2005

Rectification de deux fuites mémoire dans *sll_delete*

sll_next ne retourne plus un **int**

sll_insert, *sll_removeNex*, *sll_removeFirst*, *sll_next*, *sll_first* et *sll_last* prennent maintenant en argument un pointeur sur la liste, et ont été modifiées en conséquence

29 Juin 2005

Première publication

I - Introduction

Voici le premier tutoriel d'une future série qui traitera des structures de données en C; il s'adresse aux personnes ayant une bonne connaissance de ce langage, il est conseillé d'avoir lu un livre tel que celui de B. Kernighan et D. Ritchie : Le langage C, norme ANSI. Cette suite de tutoriels a pour but de vous fournir les bases pour comprendre l'implémentation de ces structures. Cette suite d'articles portera sur :

- Les listes simplement chaînées (c'est l'article que vous êtes en trains de lire)
- Les listes doublement chaînées
- Les piles
- Les files
- Les tables de hachage
- Les arbres
- Les graphes

Au terme de chaque tutoriel, vous disposerez d'une bibliothèque comportant les fonctions principales de manipulation de la structure de données abordée dans celui-ci.

II - Principe

La structure la plus utilisée pour manipuler des données est le tableau, qui contrairement aux listes chaînées, est implémenté de façon native dans le langage C. Cependant dans certains cas, les tableaux ne constituent pas la meilleure solution pour stocker et manipuler les données. En effet, pour insérer un élément dans un tableau, il faut d'abord déplacer tous les éléments qui sont en amont de l'endroit où l'on souhaite effectuer l'insertion, ce qui peut prendre un temps non négligeable proportionnel à la taille du tableau et à l'emplacement du futur élément. Il en est de même pour la suppression d'un élément; bien sûr ceci ne s'applique pas en cas d'ajout ou de suppression en fin de tableau. Avec une liste chaînée, le temps d'insertion et de suppression d'un élément est constant quelque soit l'emplacement de celui-ci et la taille de la liste. Elles sont aussi très pratiques pour réarranger les données, cet avantage est la conséquence directe de la facilité de manipulation des éléments.

Plus concrètement, une liste simplement chaînée est en fait constituée de maillons ayant la possibilité de pointer vers une donnée accessible et modifiable par l'utilisateur ainsi qu'un lien vers le maillon suivant. La figure suivante représente de façon schématique un élément de ce type :

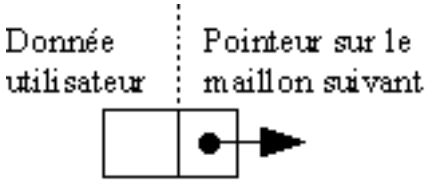

Figure 1 : représentation d'un élément d'une liste simplement chaînée.

Une liste chaînée étant une succession de maillons, dont le dernier pointe vers adresse invalide (NULL); voici une représentation possible :

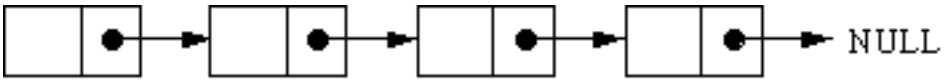

Figure 2 : représentation d'une liste chaînée en mémoire.

Bien sûr, ce n'est qu'une représentation, il se peut que les structures qui composent la liste ne soit pas placées dans l'ordre en mémoire et encore moins de façon contiguë.

Au vu de l'utilisation des listes chaînées, il se dessine clairement quelques fonctions indispensables :

- Initialisation
- Ajout d'un élément
- Suppression d'un élément
- Accès à l'élément suivant
- Accès aux donées utilisateur
- Accès au premier élément de la liste
- Accès au dernier élément de la liste
- Calcul de la taille de la liste
- Suppression de la liste entière

Le principal problème des listes simplement chaînées est l'absence de pointeur sur l'élément précédent du maillon, il est donc possible de parcourir la chaîne uniquement du début vers la fin. Pour faciliter l'implémentation, nous allons faire quelques simplifications :

- Sauvegarde du premier élément de la chaîne, ceci permet de retourner au début de la liste,
- L'ajout s'effectue à la suite de l'élément spécifié en paramètre de la fonction.

Voici donc la représentation définitive de notre structure en mémoire, *start* représentent le pointeur dédié à la sauvegarde du premier élément de la liste :

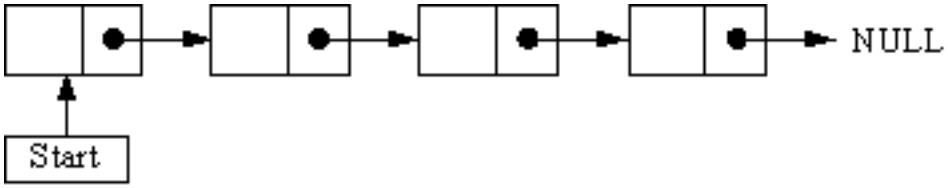

Figure 3 : organisation de la liste chaînée dans notre programme.

De plus, j'ai choisi de passer en argument, aux fonctions publiques utilisées pour manipuler des listes, un pointeur de pointeur sur la liste chaînée , il est aussi possible de passer un pointeur sur cette liste et de lui faire retourner la nouvelle liste chaînée. Cependant, j'ai préféré la première solution pour plusieurs raisons :

- Le passage par pointeur permet à la fonction appelante de modifier directement la liste,
- La valeur de retour de la fonction peut être utilisée pour d'autre chose : gestion des erreurs par exemple.
- L'élément supprimé sera celui qui suit l'élément passé en paramètre et non lui-même tout simplement parce qu'il n'est pas possible d'accéder simplement au maillon précédent pour mettre à NULL son pointeur sur l'élément suivant et actualiser le pointeur sur l'élément de fin, lors de la suppression du dernier élément.

III - La pratique

III-A - Les conventions de nommage

Dernier point avant de passer au code à proprement parler,, je vais m'attarder sur les conventions de nommage utilisés pour permettre une meilleure compréhension du code :

- Le suffixe *_s* signifie qu'il s'agit d'un tydedef de structure,
- Le préfixe *p_* est utilisé pour les noms des pointeurs,
- Les fonctions publiques de la bibliothèque commencent par le préfixe *sll_* pour **S**ingly-**L**inked **L**ists (soit listes simplement chaînées en français).

Ceci étant fixé, je vais maintenant vous présenter les différentes fonctions accompagnées des explications nécessaires pour comprendre leur fonctionnement.

III-B - La structure d'un maillon

A partir de la figure 1, la transition de cette représentation en structure au sens du langage C est simple; en effet, il nous faut un pointeur générique pour les données de l'utilisateur et un pointeur sur l'élément suivant :

```
typedef struct item
{
   struct item *next;
   void *data;
} item_s;
```

Avec :

- *next* : un pointeur sur le maillon suivant ou NULL s'il s'agit du dernier
- *data* : un pointeur générique vers la donnée stockée par l'utilisateur

Pour pouvoir sauvegarder certains éléments spécifiques de chaque liste tel que le premier élément et l'élément courant, il faut créer un nouveau type de structure :

```
typedef struct sll
{
   item_s *p_start;
   item_s *list;
} sll_s;
```

Avec :

- *p_start* : le premier élément de la liste
- *list* : l'élément courant

Il est bien sûr possible d'en ajouter d'autre, tel qu'un pointeur sur le dernier élément ou encore un compteur pour le nombre d'éléments que contient la liste.

La structure *sll_s* sera la structure à passer en paramètre à chaque fonction de la bibliothèque.

III-C - Initialisation

Cette fonction est à appeler avant toutes les autres : elle crée la structure qui va contenir la liste chaînée ainsi que le premier élément qui va servir de sentinelle.

```
sll_s *sll_new (void)
{
   sll_s *p_sll = malloc (sizeof *p_sll);

   if (p_sll)
   {
      item_s *p_l = malloc (sizeof *p_l);

      if (p_l)
      {
         p_l->data = NULL;
         p_l->next = NULL;

         p_sll->p_start = p_l;
         p_sll->list = NULL;
      }
      else
      {
         fprintf (stderr, "Memoire insufisante\n");
         exit (EXIT_FAILURE);
      }
   }
   else
   {
      fprintf (stderr, "Memoire insufisante\n");
      exit (EXIT_FAILURE);
   }
   return p_sll;
}
```

III-D - Insertion d'un élément

Cette fonction a pour but de créer un nouvel élément et de l'insérer dans la liste juste après l'élément courant de la liste. Il ne faut pas oublier de vérifier si la liste est vide (dans ce cas, *p_sll->list* est égale à NULL). Voici la schématisation correspondante, pour simplifier la sentinelle *p_start* a été omise :

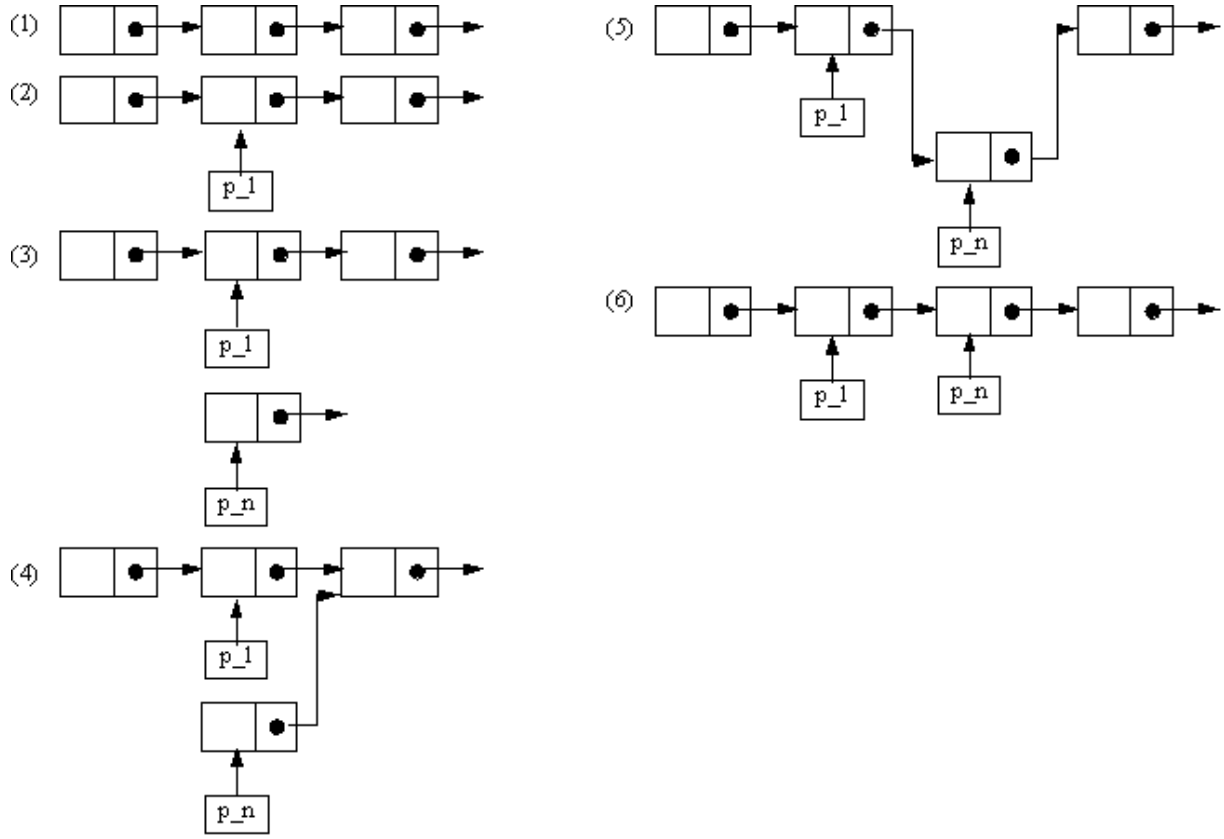

Figure 4 : représentation de l'insertion d'un élément.

1 Voici l'état de la liste avant l'appel de la fonction
2 Le pointeur *p_l* représente l'argument de la fonction, c'est donc après lui qu'il faut insérer un maillon
3 Création d'un nouvel élément pointé par *p_n*
4 On commence par faire pointer le nouvel élément sur le maillon qui suit *p_l*
5 Puis on modifie *p_l* pour que l'élément qui le suit soit celui nouvellement créé
6 Etat de la liste après l'appel de la fonction

Le plus gros du travail a été réalisé, de ce fait, le code en C ne nécessite pas de commentaire. S'il y a une erreur de mémoire la liste n'est pas modifiée :

```
void sll_insert (sll_s * p_sll, void *data)
{
   if (p_sll)
   {
/* (2) */
      item_s *p_l = p_sll->list;
      item_s *p_n = NULL;

/* (3) */
      p_n = malloc (sizeof (*p_n));
      if (p_n)
      {
         p_n->data = data;
         if (p_l == NULL)
         {
            p_sll->p_start->next = p_n;
            p_n->next = NULL;
         }
         else
         {
/* (4) */
            p_n->next = p_l->next;
```

```
   /* (5) */
            p_l->next = p_n;
         }
   /* (6) */
         p_sll->list = p_n;
      }
      else
      {
         fprintf (stderr, "Memoire insuffisante\n");
         exit (EXIT_FAILURE);
      }
   }
}
```

III-E - Suppression d'un élément

Après avoir ajouté un élément, il est intéressant de voir comment en enlever. Comme spécifié dans le paragraphe exposant le principe, c'est l'élément suivant celui spécifié en paramètre qui est supprimé, par conséquent on ne peut pas passer le dernier élément de la liste à cette fonction, et il est aussi impossible de supprimer le premier élément de cette manière. Voici la schématisation correspondante, pour simplifier *p_start* est toujours absent du schéma :

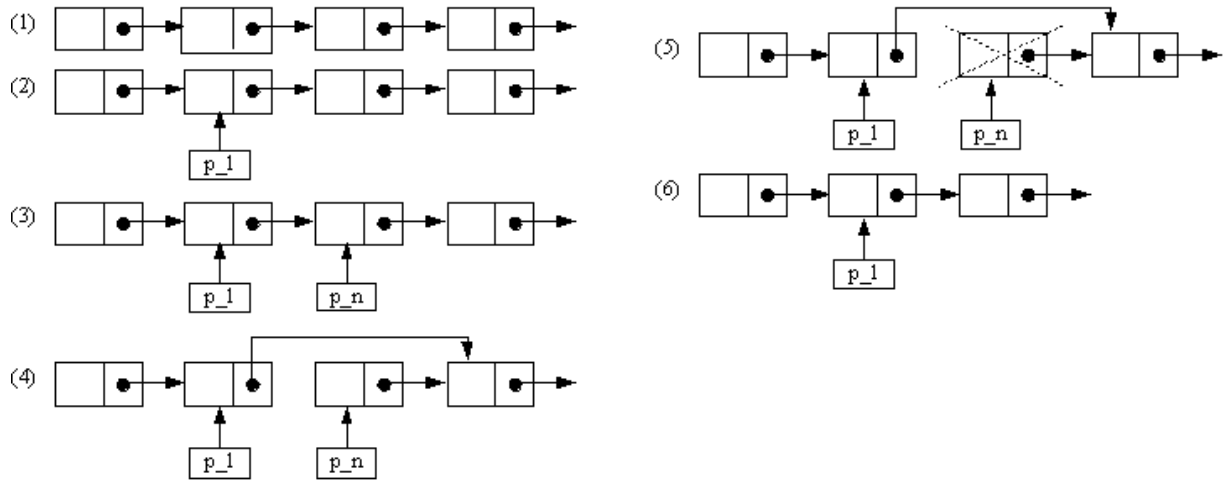

Figure 5 : représentation de la suppression d'un élément.

1 Voici l'état de la liste avant l'appel de la fonction
2 Le pointeur *p_l* représente l'argument de la fonction, c'est donc l'élément suivant qu'il faut supprimer
3 Sauvegarde de l'élément à supprimer grâce au pointeur *p_n*
4 Modification du pointeur sur l'élément suivant pour qu'il pointe sur l'élément qui suit immédiatement celui à supprimer
5 Libération de la mémoire
6 État de la liste après l'appel de la fonction

Maintenant passons au code :

```
void sll_removeNext (sll_s * p_sll)
{
   if (p_sll && p_sll->list)
   {
/* (2) */
      item_s *p_l = p_sll->list;
      item_s *p_n = NULL;

/* (3) */
      p_n = p_l->next;
```

```
   /* (4) */
         p_l->next = p_n->next;
   /* (5) */
         free (p_n);
         p_n = NULL;
      }
   }
```

III-F - Suppression du premier élément de la liste

Avec la fonction *sll_removeNext*, il est impossible de supprimer le premier élément de la liste puisque l'utilisateur n'a pas accès à la sentinelle, c'est pourquoi on est obligé de créer une fonction spéciale qui va supprimer cet élément en passant la sentinelle à la fonction *sll_removeNext* :

```
   void sll_removeFirst (sll_s * p_sll)
   {
      if (p_sll)
      {
         p_sll->list = p_sll->p_start;
         sll_removeNext (p_sll);
      }
   }
```

III-G - Accès à l'élément suivant

Cette fonction se contente de retourner l'élément suivant dans la liste, en faisant tout de même attention à la valeur de *p_l* :

```
   void sll_next (sll_s * p_sll)
   {
      if (p_sll && p_sll->list)
      {
         p_sll->list = p_sll->list->next;
      }
   }
```

III-H - Accés aux données stockées

Les listes chaînées permettent à l'utilisateur de stocker des données, il est donc normal qu'il puisse y accéder :

```
   void *sll_data (sll_s * p_sll)
   {
      return ((p_sll && p_sll->list) ? p_sll->list->data : NULL);
   }
```

III-I - Accès au premier élément de la liste

Cette fonction est rendue extrêmement simple par l'existence de la sentinelle :

```
   void sll_first (sll_s * p_sll)
   {
      if (p_sll)
      {
         p_sll->list = p_sll->p_start->next;
      }
   }
```

III-J - Accès au dernier élément de la liste

Contrairement à la fonction précédente, on n'a pas de sentinelle pointant sur le dernier élément, par conséquent, pour le retrouver, il faut parcourir la liste jusqu'à tomber sur le pointeur NULL :

```
vvoid sll_last (sll_s * p_sll)
{
   if (p_sll)
   {
      while (p_sll->list->next != NULL)
      {
         sll_next (p_sll);
      }
   }
}
```

III-K - Calcul de la taille de la liste

Cette fonction retourne le nombre d'éléments présents dans la liste et ce en partant du début de la liste et en incrémentant le nombre d'éléments jusqu'à arriver à la fin de la liste en utilisant la propriété de la fonction *sll_next* qui retourne NULL en fin de liste :

```
size_t sll_sizeof (sll_s * p_sll)
{
   size_t n = 0;

   if (p_sll)
   {
      sll_first (p_sll);
      while (p_sll->list != NULL)
      {
         n++;
         sll_next (p_sll);
      }
   }
   return n;
}
```

III-L - Suppression de la liste

Voici la dernière fonction de notre bibliothèque, il s'agit d'une fonction permettant de détruire entièrement la liste en commençant par le premier élément et de supprimer chaque élément jusqu'au dernier, ou plus précisément jusqu'à ce que le premier élément devienne le dernier puis celui-ci est supprimé manuellement.

```
void sll_delete (sll_s ** pp_sll)
{
   if (pp_sll && *pp_sll)
   {
      sll_first (*pp_sll);

      while ((*pp_sll)->list->next != NULL)
      {
         sll_removeNext (*pp_sll);
      }
      sll_removeFirst (*pp_sll);
      free ((*pp_sll)->list);
      (*pp_sll)->list = NULL;
      free (*pp_sll), *pp_sll = NULL;
   }
}
```

IV - Conclusion

Il existe bien d'autres fonctions à coder mais cet exemple se veut léger (j'utilise la version doublement chaînée pour créer un script CGI universel donc il est important que le programme résultant soit le plus léger possible pour améliorer son utilisation via l'internet). Cependant si vous souhaitez compléter cette bibliothèque, ce qui est un très bon exercice, vous pouvez vous inspirer des fonctions présentent dans la glib. Voilà, c'est la fin de ce premier article d'une longue série, en espérant vous avoir transmis ma passion pour le C au travers de celui-ci et aussi vous donner envie de lire la suite.

V - Autres documentations

Liste simplement chaînée par CGI

VI - Remerciements

Un grand merci à Emmanuel Delahaye pour ses conseils et les corrections qu'il a apporté au code de la bibliothèque

Merci à Anomaly et à MD Software pour leur relecture attentive de cet article

VIII - Code source complet

Voici le code complet de la bibliothèque (téléchargez l'archive zippée) avec un fichier *main.c* pour illustrer son utilisation, pour ne pas manquer à la tradition, il s'agit d'un programme qui affiche "Hello world!" en utilisant la plupart des fonctions vues précédemment.

listesimple.h

```
#ifndef H_LISTESIMPLE
#define H_LISTESIMPLE

#include <stddef.h>             /* pour size_t */

typedef struct sll sll_s;

sll_s *sll_new (void);
void sll_insert (sll_s *, void *);
void sll_removeNext (sll_s *);
void sll_removeFirst (sll_s *);
void sll_next (sll_s *);
void *sll_data (sll_s *);
void sll_first (sll_s *);
void sll_last (sll_s *);
size_t sll_sizeof (sll_s *);
void sll_delete (sll_s **);

#endif /* not H_LISTESIMPLE */
```

listesimple.c

```
#include <stdio.h>
#include <stdlib.h>
#include "listesimple.h"

typedef struct item
{
   struct item *next;
   void *data;
} item_s;

struct sll
{
   item_s *p_start;
   item_s *list;
};

sll_s *sll_new (void)
{
   sll_s *p_sll = malloc (sizeof *p_sll);

   if (p_sll)
   {
      item_s *p_l = malloc (sizeof *p_l);

      if (p_l)
      {
         p_l->data = NULL;
         p_l->next = NULL;

         p_sll->p_start = p_l;
         p_sll->list = NULL;
      }
      else
      {
         fprintf (stderr, "Memoire insufisante\n");
         exit (EXIT_FAILURE);
      }
   }
   else
   {
      fprintf (stderr, "Memoire insufisante\n");
      exit (EXIT_FAILURE);
   }
   return p_sll;
}
```

listesimple.c

```
void sll_insert (sll_s * p_sll, void *data)
{
   if (p_sll)
   {
      item_s *p_l = p_sll->list;
      item_s *p_n = NULL;

      p_n = malloc (sizeof (*p_n));
      if (p_n)
      {
         p_n->data = data;
         if (p_l == NULL)
         {
            p_sll->p_start->next = p_n;
            p_n->next = NULL;
         }
         else
         {
            p_n->next = p_l->next;
            p_l->next = p_n;
         }
         p_sll->list = p_n;
      }
      else
      {
         fprintf (stderr, "Memoire insuffisante\n");
         exit (EXIT_FAILURE);
      }
   }
}

void sll_removeNext (sll_s * p_sll)
{
   if (p_sll && p_sll->list)
   {
      item_s *p_l = p_sll->list;
      item_s *p_n = NULL;

      p_n = p_l->next;
      p_l->next = p_n->next;
      free (p_n);
      p_n = NULL;
   }
}

void sll_removeFirst (sll_s * p_sll)
{
   if (p_sll)
   {
      p_sll->list = p_sll->p_start;
      sll_removeNext (p_sll);
   }
}

void sll_next (sll_s * p_sll)
{
   if (p_sll && p_sll->list)
   {
      p_sll->list = p_sll->list->next;
   }
}

void *sll_data (sll_s * p_sll)
{
   return ((p_sll && p_sll->list) ? p_sll->list->data : NULL);
}

void sll_first (sll_s * p_sll)
{
   if (p_sll)
   {
      p_sll->list = p_sll->p_start->next;
   }
}

void sll_last (sll_s * p_sll)
{
```

listesimple.c

```
    if (p_sll)
    {
       while (p_sll->list->next != NULL)
       {
          sll_next (p_sll);
       }
    }
}

size_t sll_sizeof (sll_s * p_sll)
{
   size_t n = 0;

   if (p_sll)
   {
      sll_first (p_sll);
      while (p_sll->list != NULL)
      {
         n++;
         sll_next (p_sll);
      }
   }
   return n;
}

void sll_delete (sll_s ** pp_sll)
{
   if (pp_sll && *pp_sll)
   {
      sll_first (*pp_sll);

      while ((*pp_sll)->list->next != NULL)
      {
         sll_removeNext (*pp_sll);
      }
      sll_removeFirst (*pp_sll);
      free ((*pp_sll)->list);
      (*pp_sll)->list = NULL;
      free (*pp_sll), *pp_sll = NULL;
   }
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "listesimple.h"

static void list_print (sll_s *);

int main (void)
{
   sll_s *p_sll = NULL;
   char text[][6] = { "Hello", " ", "World", "!" };

   printf ("\t-- Initialisation --\n");
   p_sll = sll_new ();
   list_print (p_sll);

   printf ("\n\t-- Insertion --\n");
   {
      int i;

      for (i = 0; i < 4; i++)
      {
         sll_insert (p_sll, text[i]);
      }
   }
   list_print (p_sll);

   printf ("\n\t-- Suppression --\n");
   sll_removeFirst (p_sll);
   list_print (p_sll);

   printf ("\n\t-- Destruction --\n");
   sll_delete (&p_sll);
   list_print (p_sll);
```

main.c

```
    return EXIT_SUCCESS;
}

static void list_print (sll_s * p_l)
{
    int i;
    int size = sll_sizeof (p_l);

    sll_first (p_l);
    for (i = 0; i < size; i++)
    {
        if (sll_data (p_l) != NULL)
        {
            printf ("%s", (char *) sll_data (p_l));
        }
        sll_next (p_l);
    }
    printf ("\n");
}
```