Programmation impérative

Avancé
Cahier

Licence informatique 2ème année

Université de La Rochelle

<christophe.demko@univ-lr.fr>

La Rochelle Université

Ce cahier illustre les structures de données classiques en langage C :

— les collections dynamiques

— les listes

Les lignes commençant par $ représente une commande *unix*. Le $ ne fait pas partie de la commande proprement dite mais symbolise l'invite de commande.

Table des matières

Table des figures

Liste des exercices

1 Collections dynamiques à une dimension

Dans cet ensemble d'exercices, nous vous demandons d'écrire un ensemble de fonctions permettant de gérer les collections dynamiques à une dimension en respectant les contraintes suivantes (voir figure 1) :

- les collections peuvent contenir n'importe quel type de données ;
- l'accès à l'élément `i` de la collection dynamique a se fait par la construction `a[i]` ;
- le nombre d'éléments alloués peut être supérieur au nombre d'éléments réel ;
- soit `max` le nombre d'éléments alloués et `length` le nombre d'éléments réel; les algorithmes utilisés doivent toujours conserver la relation `max/ratio <= length <= max` (`ratio` doit toujours être supérieur ou égal à 1). L'astuce consiste à réallouer la mémoire utilisée par la collection lorsque la condition n'est plus vrai en utilisant un nouveau maximum qui permettent de rétalir la relation `max/ratio <= length <= max`
  - soit `new_length*2*ratio/(ratio+1)` de telle manière que la moyenne de `new_max/ratio` et `new_max` soit égale à `new_length` ;
  - soit `sqrt(ratio)*new_length` de telle manière que les rapports `new_length/(new_max/ratio)` et `new_max/new_length` soient égaux.

— 2 fonctions doivent pouvoir donner le nombre d'éléments actuel de la collection `array_get_length(a)` et la taille en octets d'un élément de la collection `array_get_size(a)`;
— il doit être possible de changer la taille de ces collections de façon dynamique et transparente pour l'utilisateur par une fonction
`extern void *array_set_length(void *array, unsigned int new_length);`.

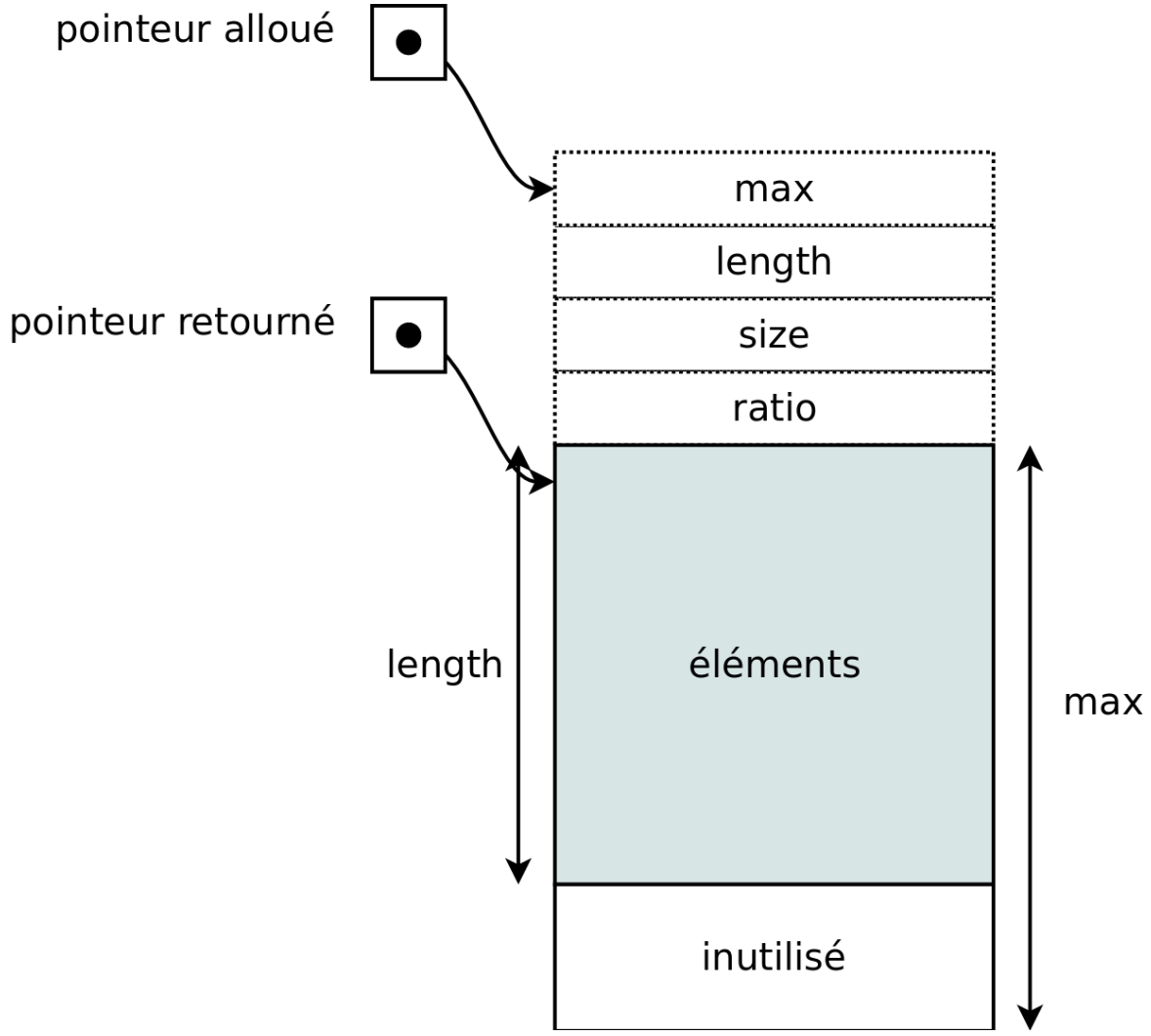

FIGURE 1 – Représentation d'une collection dynamique

1.1 Opérations sur les collections dynamiques

Exercice 1 *(Définition du type `Array`)*
Dans un fichier `array.c`, définissez le type `Array` permettant de représenter l'ensemble des informations de la collection dynamique.

```
typedef struct _Array Array;
```

Ni le type, ni la structure ne seront présents dans le fichier `array.h` mais pourront être présents dans un fichier `array.inc`.

Exercice 2 *(Fonctions d'initialisation et de finalisation sur les collections dynamiques)*

1. Écrivez deux fonctions permettant d'initialiser (resp. de finaliser) la librairie gérant les collections dynamiques.

   ```
   extern bool array_init(void);
   extern bool array_finish(void);
   ```

   Les 2 fonctions doivent simplement afficher un message.

2. Écrivez un programme de test `test-init-finish` permettant de valider les 2 fonctions écrites.

Exercice 3 *(Fonctions de création et de destruction sur les collections dynamiques)*

1. Écrivez une fonction permettant de créer par allocation une nouvelle collection dynamique.

   ```
   extern void *array_create_full(
     const unsigned int size,
     const unsigned int length,
     double ratio
   );
   ```

   — `size` représente la taille en octets d'un élément ;
   — `length` représente la longueur initiale de la collection ;
   — `ratio` est un nombre exprimant le compromis entre la taille réellement utilisée et celle en attente (il est initialisé à 1 si valeur passée en paramètre est inférieure à 1).

   Au retour de l'appel de cette fonction, la collection dynamique devra avoir été créé. Si un problème d'allocation survient, la fonction doit retourner `NULL`.

2. Écrivez une fonction permettant de créer par allocation une nouvelle collection dynamique.

   ```
   extern void *array_create_default(const unsigned int size);
   ```

   Cet appel dit être équivalent à

   ```
   array_create_full(size, array_default_length, array_default_ratio);
   ```

   où

   — `array_default_length` est la longueur par défaut des collections (0 au départ).
   — `array_default_ratio` est le ratio par défaut des collections (2.0 au départ).

3. Écrivez une fonction permettant de libérer toute la mémoire utilisée par une collection dynamique.

   ```
   extern void array_destroy(void * array);
   ```

4. Écrivez un programme de test `test-create-destroy` permettant de valider les fonctions écrites.

Exercice 4 *(Fonctions d'accès aux données sur les collections dynamiques)*

1. Écrivez trois fonctions permettant de récupérer la taille d'un élément, la longueur et le ratio d'une collection dynamique.

   ```
   extern unsigned int array_get_length(const void *array);
   extern unsigned int array_get_size(const void *array);
   extern double array_get_ratio(const void *array);
   ```

Écrivez une fonction permettant de modifier la longueur d'une collection dynamique.

```
extern void *array_set_length(void *array, const unsigned int
↪  new_length);
```

2. Écrivez deux programmes de test `test-get` et `test-set` permettant de valider les 4 fonctions écrites.

Exercice 5 *(Insertion de valeurs dans une collection dynamique)*
Écrivez 6 fonctions permettant d'insérer des valeurs dans une collection dynamique.

```
extern void *array_insert_values(
  void *dest,
  const void *src,
  const unsigned int index,
  const unsigned int length
);
extern void *array_insert_value(
  void *dest,
  const void *src,
  const unsigned int index
);

extern void *array_prepend_values(
  void *dest,
  const void *src,
  const unsigned int length
);
extern void *array_prepend_value(
  void *dest,
  const void *src
);

extern void *array_append_values(
  void *dest,
  const void *src,
  const unsigned int length
);
extern void *array_append_value(
  void *dest,
  const void *src
);
```

- `array_insert_value(dest, src, index)` doit être équivalent à `array_insert_values(dest, src, index, 1)`
- `array_append_values(dest, src, length)` doit être équivalent à `array_insert_values(dest, src, array_get_length(dest), length)`
- `array_append_value(dest, src)` doit être équivalent à `array_append_values(dest, src, 1)`
- `array_prepend_values(dest, src, length)` doit être équivalent à

`array_insert_values(dest, src, 0, length)`
— `array_prepend_value(dest, src)` doit être équivalent à `array_prepend_values(dest, src, 1)`

Dans ces fonctions,
— `length` représente le nombre d'éléments à insérer à partir de `src`;
— `index` représente l'endroit où insérer les éléments dans `dest`;

Exercice 6 *(Suppression de valeurs dans une collection dynamique)*
Écrivez 2 fonctions permettant de supprimer des valeurs dans une collection dynamique.

```
extern void *array_remove_values(
  void *array,
  const unsigned int index,
  const unsigned int length
);
extern void *array_remove_value(
  void *array,
  const unsigned int index
);
```

— `array_remove_value(array, index)` doit être équivalent à `array_remove_values(array, index, 1)`

Dans ces fonctions,
— `length` représente le nombre d'éléments à supprimer de `array`;
— `index` représente l'endroit où commencer la suppression;

Exercice 7 *(Conversion en chaîne de caractères des collections dynamiques)*
Écrire une fonction

```
extern const char* array_to_string(
  const void* array,
  const char *(*to_string)(const void*)
);
```

permettant de convertir une collection dynamique en chaîne de caractères. La chaine produite devra commencer par le caractère "`[`", être terminée par le caractère "`]`" et séparera la conversion de chaque élément par le caractère "`,`". La conversion de chaque élément sera assurée par la fonction pointée par le paramètre `to_string` qui recevra en argument chaque élément de la collection.

1.2 Itérations sur les collections dynamiques

Il peut être intéressant d'essayer d'implémenter le *patron* de programmation *Itérateur* sur les collections dynamiques.

Exercice 8 *(Définition du type `ArrayIterator`)*
Définissez un nouveau type `ArrayIterator` dans les fichiers `array.h` (pour le type), `array.c` (pour le type) et `array.inc` (pour la structure) permettant d'itérer sur une collection dynamique.

Ce type contiendra un pointeur vers la collection itérée et un entier indiquant quel indice l'itérateur a atteint.

Exercice 9 *(Fonctions de création et de destruction des itérateurs sur les collections dynamiques)*
Écrivez deux fonctions

```
extern ArrayIterator *array_iterator_create(const void *array);
extern void array_iterator_destroy(ArrayIterator * iterator);
```

permettant de créer un itérateur et de libérer sa mémoire.

Exercice 10 *(Fonctions d'itération sur les collections dynamiques)*
Écrivez trois fonctions

```
extern ArrayIterator *array_iterator_next(ArrayIterator * iterator);
extern ArrayIterator *array_iterator_rewind(ArrayIterator * iterator);
extern bool array_iterator_valid(const ArrayIterator * iterator);
```

permettant d'itérer à partir d'un itérateur.

- `array_iterator_next` permet de faire avancer l'itérateur à la prochaine valeur ;
- `array_iterator_rewind` permet de réinitialiser l'itérateur sur la première valeur ;
- `array_iterator_valid` permet de savoir s'il reste encore des éléments sur lesquels itérer dans l'itérateur ;

Exercice 11 *(Fonctions d'accès aux données des itérateurs sur les collections dynamiques)*
Écrivez une fonction

```
extern void *array_iterator_get(const ArrayIterator * iterator);
```

permettant d'accéder à l'élément courant d'un itérateur.

2 Listes génériques

Une liste est une collection ordonnée d'objets correspondant à la définition formelle suivante : une liste peut être

- soit vide ;
- soit composée d'un élément et du reste de la liste. Le reste de la liste est une liste lui-même.

C'est une définition récursive qui imposera certains choix lors de l'implémentation.

L'avantage principal d'une structure de type *liste*: l'ajoût d'élément en tête de liste est en *O(1)*.

L'inconvénient principal d'une structure de type *liste*: l'accès à l'élément numéro *i* de la liste est en moyenne en *O(n)* (n étant le nombre d'éléments de la liste).

Dans les exercices qui suivent, vous vous efforcerez de pratiquer la politique *DRY* (*Don't Repeat Yourself*) en factorisant votre code au maximum.

2.1 Opérations sur les listes génériques

Exercice 12 *(Définition du type `List`)*
Dans un fichier `list.h`, définir le type `List` permettant de représenter les listes génériques. Le type de données porté par les éléments d'une liste générique sera représenté par le type `void *` du langage C.

Exercice 13 *(Fonctions d'initialisation et de terminaison sur les listes)*

1. Dans un fichier `list.c`, écrivez deux fonctions permettant d'initaliser (resp. de terminer) la librairie gérant les listes.

   ```
   extern bool list_init(void);
   extern bool list_finish(void);
   ```

   Les 2 fonctions doivent pour l'instant simplement afficher un message.

2. Écrivez un programme de test `test-init-finish` permettant de valider les 2 fonctions écrites.

Exercice 14 *(Fonction d'ajoût en début de liste)*
Dans un fichier `list.c`, écrire une fonction

```
extern List *list_prepend(List * list, const void *data);
```

permettant d'ajouter l'élément `data` au début de la liste `list` (voir figure 2).

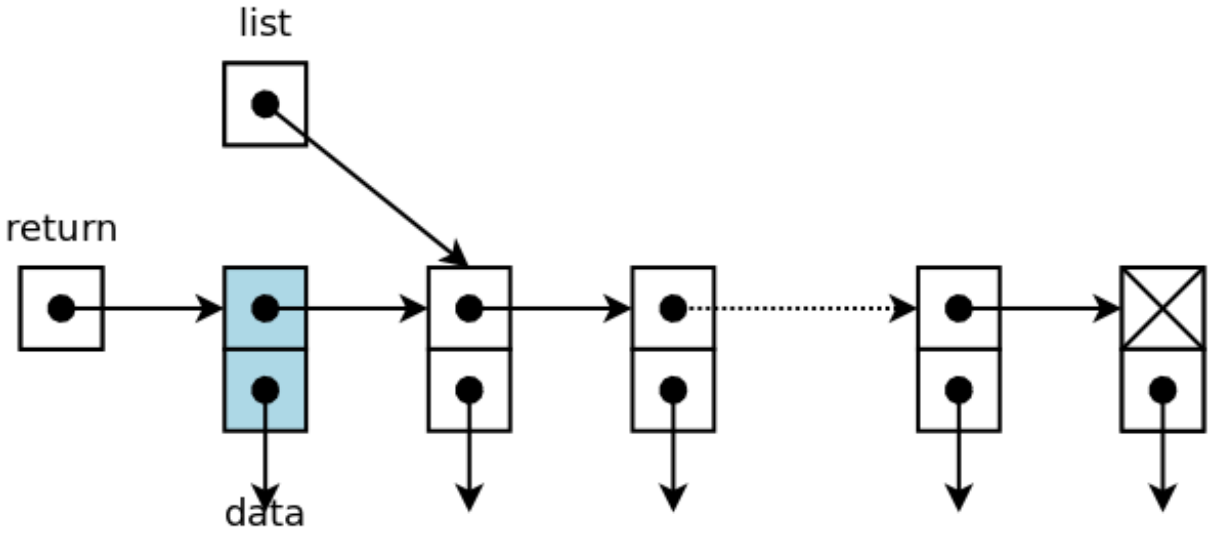

FIGURE 2 – Ajoût d'un élément au début d'une liste

Exercice 15 *(Fonction d'ajoût en fin de liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_append(List * list, const void *data);
```

permettant d'ajouter l'élément `data` à la fin de la liste `list` (voir figure 3).

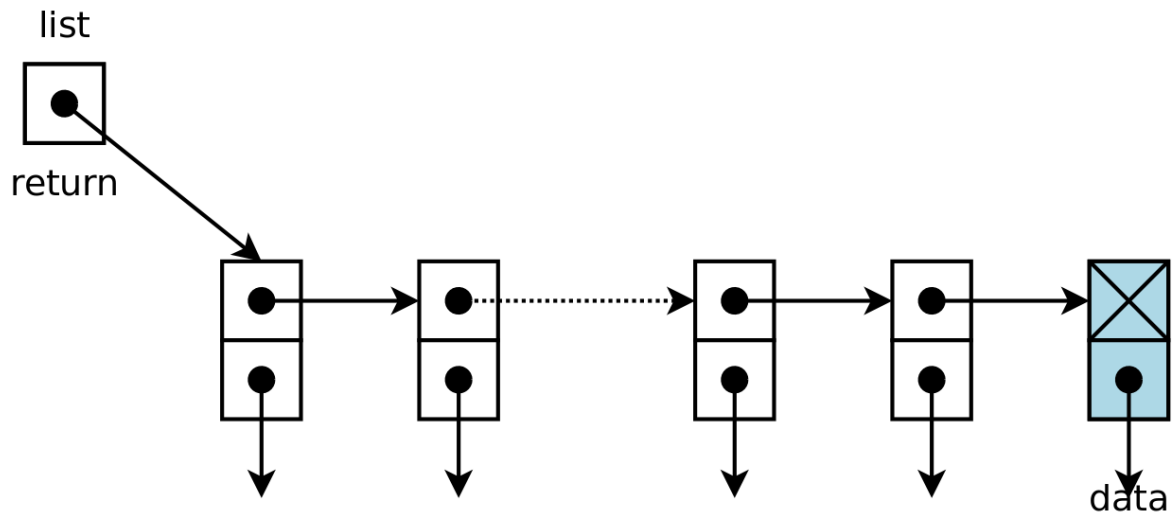

FIGURE 3 – Ajoût d'un élément à la fin d'une liste

Exercice 16 *(Fonction de destruction de liste)*
Dans un fichier `list.c`, écrire une fonction

```
extern void list_destroy(List * list);
```

permettant de libérer l'ensemble de la mémoire allouée pour la liste `list` (voir figure 4).

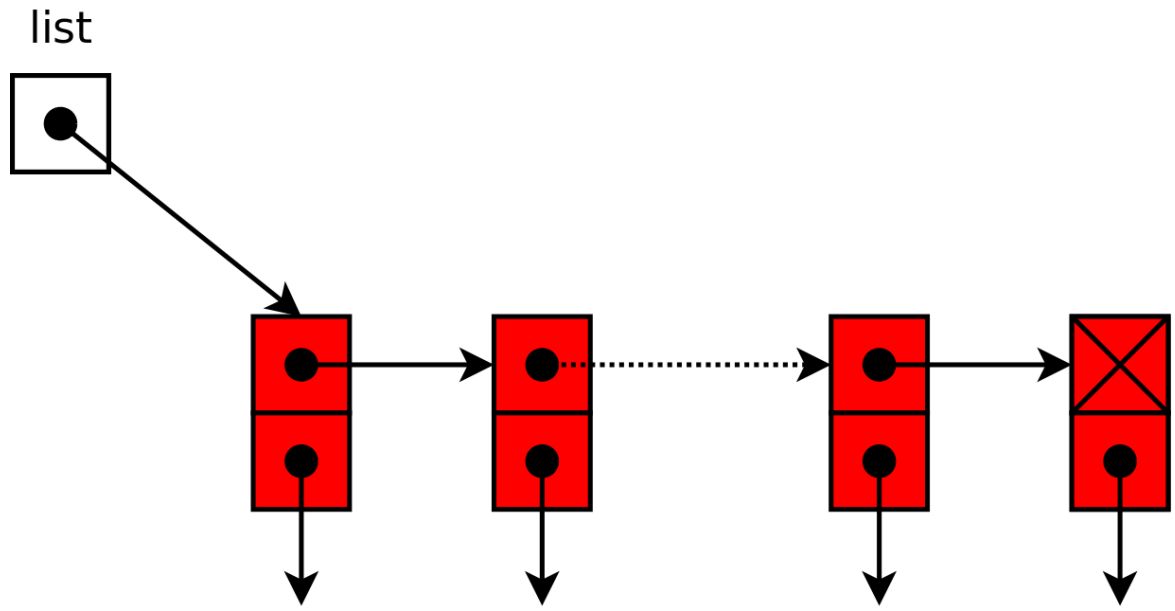

FIGURE 4 – Destruction d'une liste

Exercice 17 *(Fonction de concaténation de listes)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_concat(List * dest, const List * src);
```

permettant d'ajouter la liste `src` à la fin de `dest` (voir figure 5).

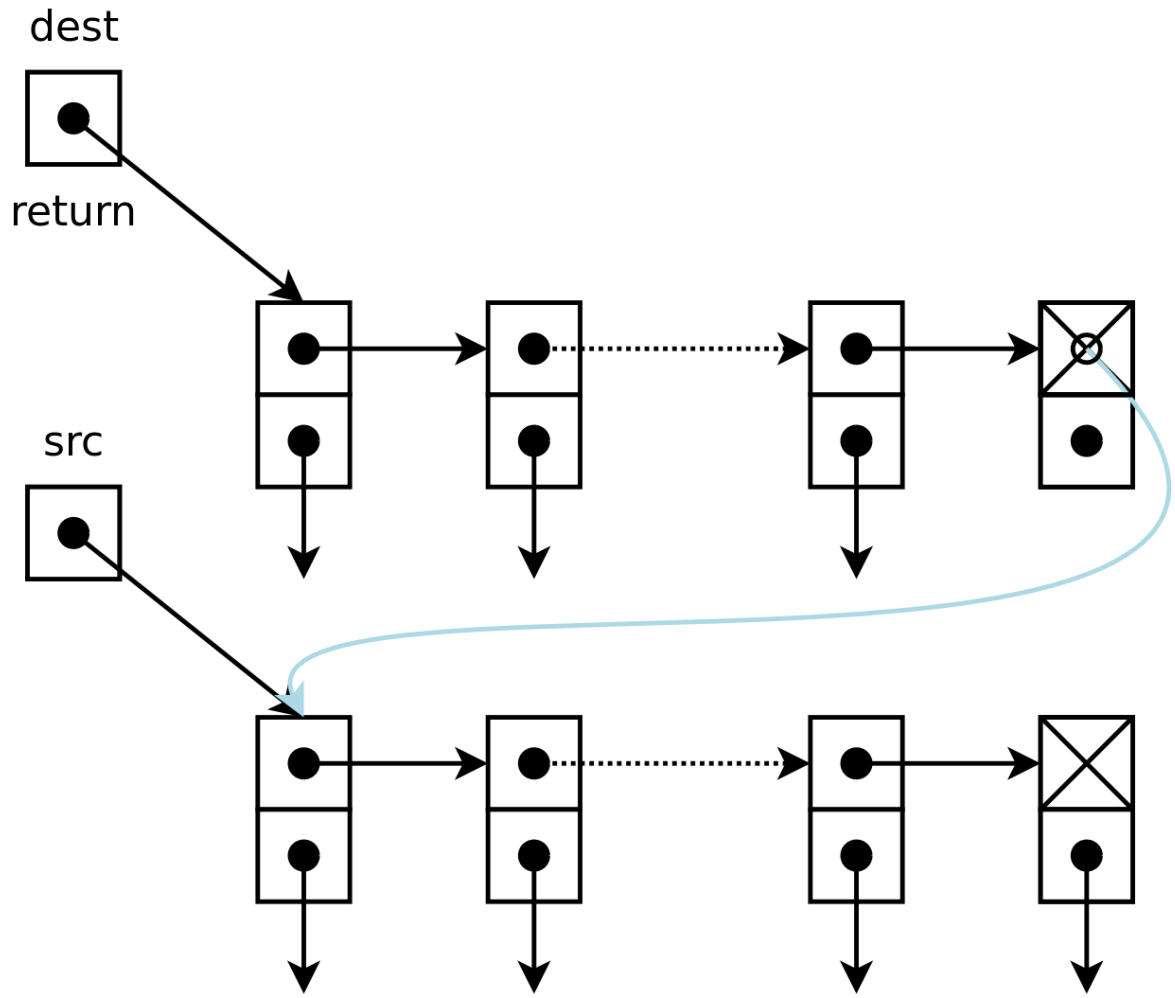

FIGURE 5 – Concaténation de listes

Exercice 18 *(Calcul de la longueur d'une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern unsigned int list_length(const List * list);
```

permettant de calculer la longueur d'une liste.

Exercice 19 *(Accès au $n^{ème}$ élément d'une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern void *list_nth(const List * list, const unsigned int nth);
```

permettant de récupérer l'élément numéro `nth`. La valeur `NULL` sera retournée si `nth` est supérieur ou égal à la longueur de la liste. Les éléments sont considérés numérotés de `0` à `list_length(list)-1`.

Exercice 20 *(Recherche de l'index d'un élément d'une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern int list_index(const List * list, const void *data);
```

permettant de connaître l'index de la donnée passée en argument. Si elle n'est pas trouvée, le nombre `-1` est retourné.

Exercice 21 *(Insertion avant un élément dans une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_insert_before(
  List * list,
  const void *before,
  const void *data
);
```

permettant d'insérer la donnée `data` avant la donnée `before`. Si la donnée `before` n'existe pas dans la liste, la nouvelle donnée sera insérée au début (voir figure 6).

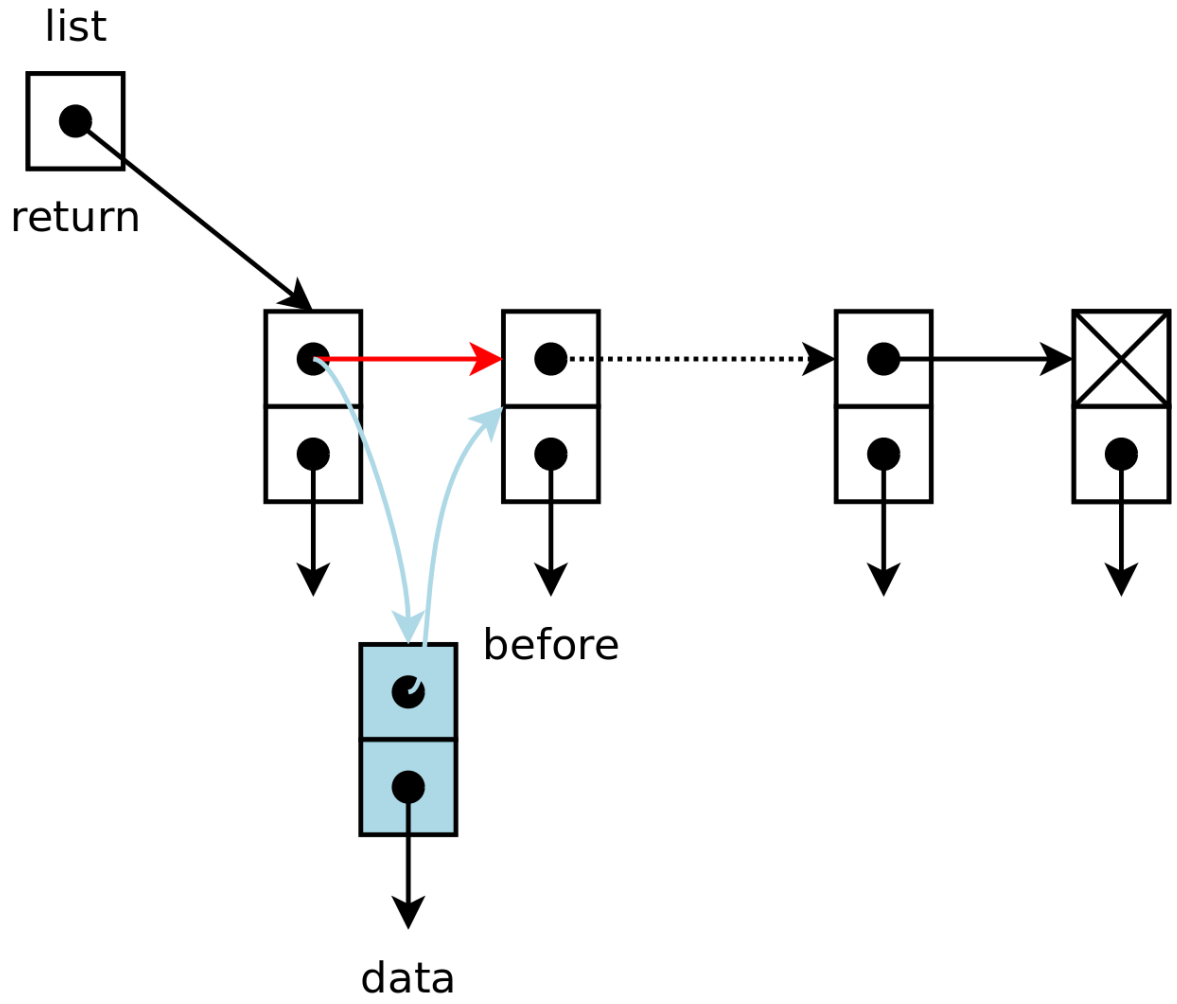

FIGURE 6 – Insertion avant un élément dans une liste

Exercice 22 *(Insertion après un élément dans une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_insert_after(
  List * list,
```

```
  const void *after,
  const void *data
);
```

permettant d'insérer la donnée `data` après la donnée `after`. Si la donnée `after` n'existe pas dans la liste, la nouvelle donnée sera insérée à la fin (voir figure 7).

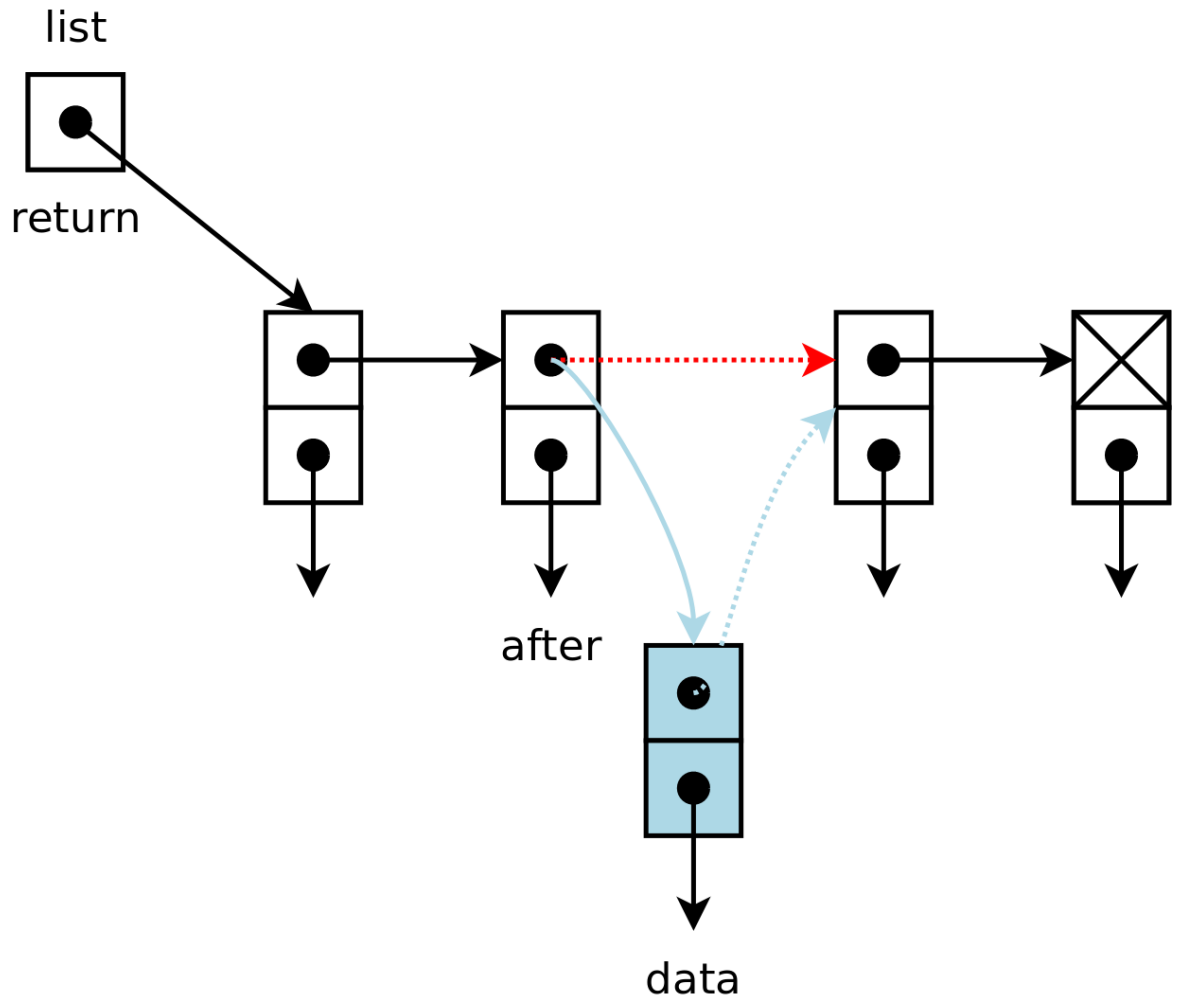

FIGURE 7 – Insertion après un élément dans une liste

Exercice 23 *(Insertion à la $n^{ème}$ position dans une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_insert_nth(
  List * list,
  const unsigned int nth,
  const void *data
);
```

permettant d'insérer la donnée `data` en position `nth`. Si `nth` dépasse la longueur de la liste, la donnée sera insérée à la fin (voir figure 8).

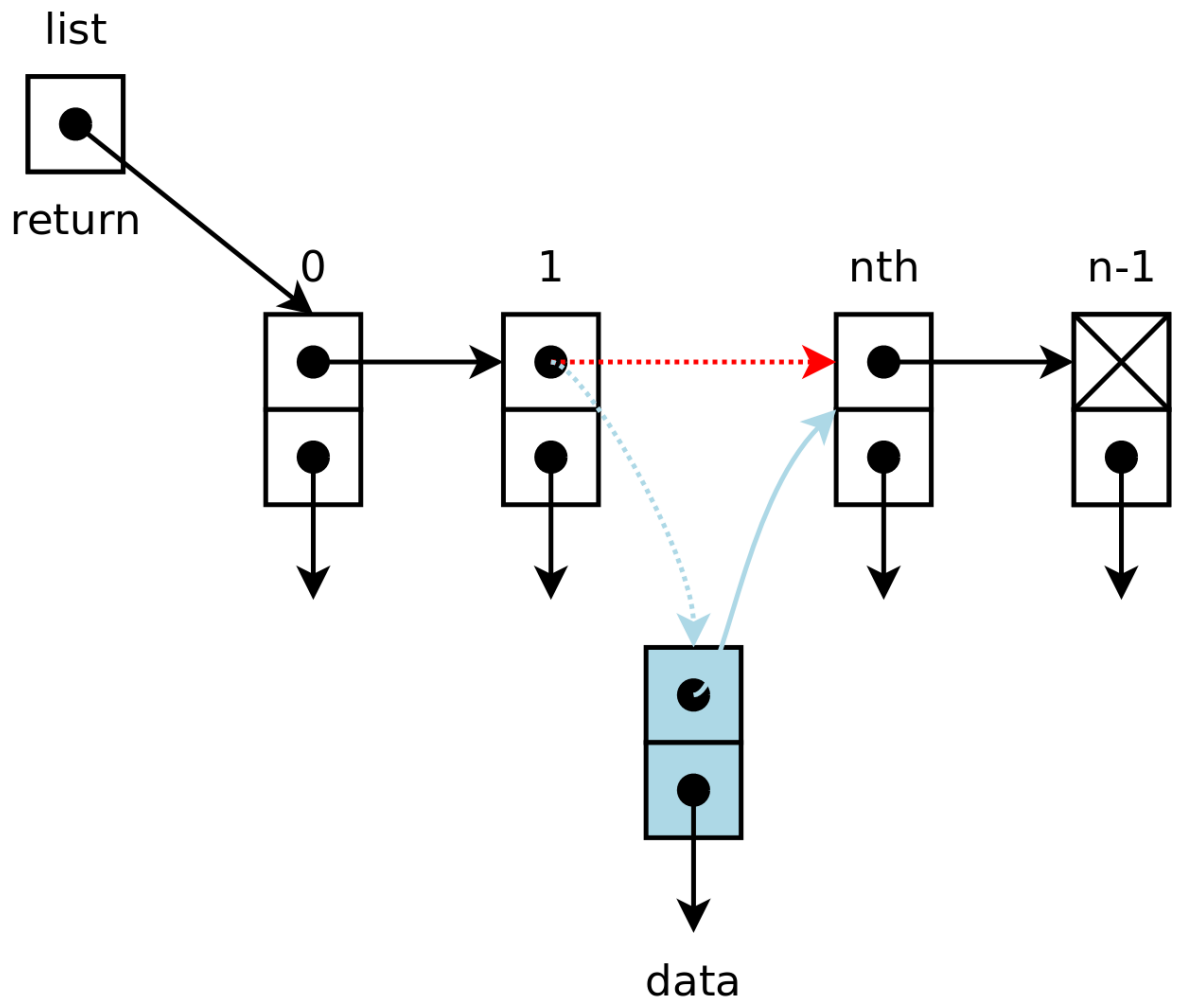

FIGURE 8 – Insertion à la n^{ème} position dans une liste

Exercice 24 *(Suppression d'une donnée dans une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_remove_data(List * list, const void *data);
```

permettant de retirer la donnée `data` de la liste. Si la donnée n'est pas trouvée, la liste retournée est inchangée (voir figure 9).

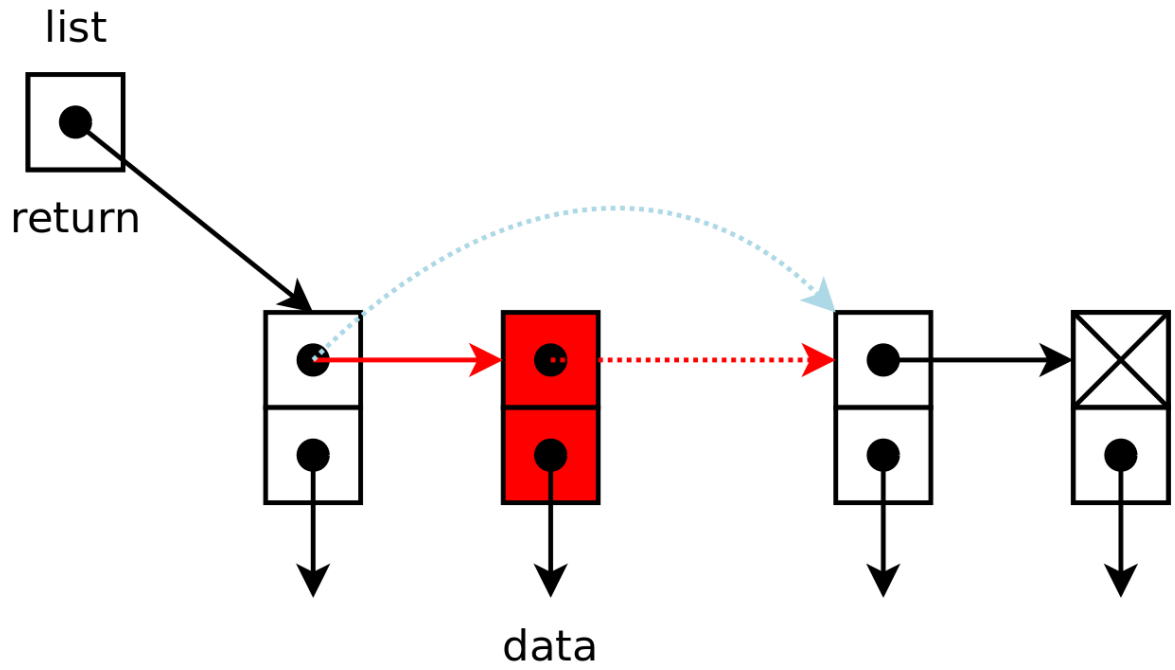

FIGURE 9 – Suppression d'une donnée dans une liste

Exercice 25 *(Suppression de la n^{ème} donnée dans une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_remove_nth(List * list, const unsigned int nth);
```

permettant de retirer la donnée numéro `nth` de la liste. Si `nth` dépasse la longueur de la liste, la liste retournée est inchangée (voir figure 10).

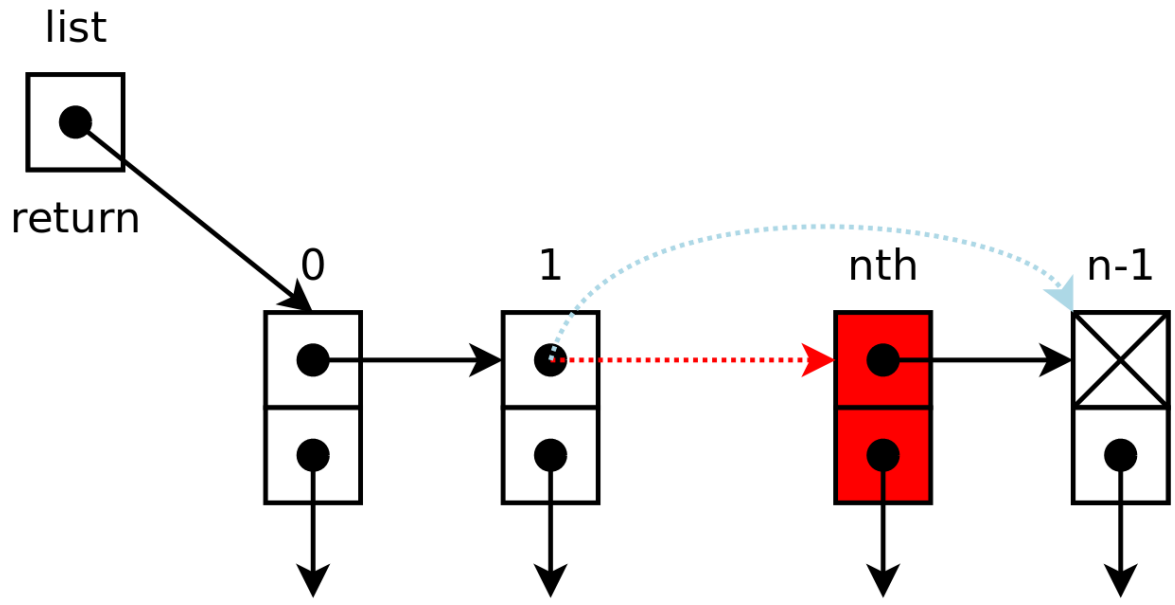

FIGURE 10 – Suppression de la n^{ème} donnée dans une liste

Exercice 26 *(Clonage d'une liste)*
Dans le fichier `list.c`, écrire une fonction

```
extern List *list_clone(List * list);
```

permettant de cloner une liste.

Exercice 27 *(Inversion des listes)*
Écrire une fonction **itérative** permettant d'inverser une liste ; i.e. le dernier élément devient le premier, l'avant-dernier le deuxième, etc… Il n'y doit pas y avoir de création de nouveaux éléments de liste.

```
extern List *list_reverse(List * list);
```

La fonction retourne la liste inversée.

Exercice 28 *(Tri d'une liste)*

1. Dans le fichier `list.c`, écrire une fonction

   ```
   extern List *list_quick_sort(
     List * list,
     int (*compare) (const void *, const void *)
   );
   ```

   permettant de trier une liste par un tri à pivot.

2. Dans le fichier `list.c`, écrire une fonction

   ```
   extern List *list_merge_sort(
     List * list,
     int (*compare) (const void *, const void *)
   );
   ```

   permettant de trier une liste par un tri à fusion.

3. Dans le fichier `list.c`, écrire une fonction

   ```
   extern List *list_bubble_sort(
     List * list,
     int (*compare) (const void *, const void *)
   );
   ```

   permettant de trier une liste par un tri à bulles.

4. Écrire un programme permettant de tester la rapidité des méthodes de tri.

Exercice 29 *(Conversion en chaines de caractères d'une liste)*
Écrire une fonction

```
extern const char* list_to_string(
  const List * list,
  const char *(*to_string)(const void*)
);
```

permettant de convertir une liste en chaîne de caractères. La chaine produite devra commencer par le caractère '[', être terminée par le caractère ']' et séparera la conversion de chaque élément par le

caractère ‘,’. La conversion de chaque élément sera assurée par la fonction pointée par le paramètre `to_string` qui recevra en argument l’élément à convertir.
Par exemple, le programme de test suivant :

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

#ifdef NDEBUG
#undef NDEBUG
#endif
#include <assert.h>

#include "./list.h"

const char *int_to_string(const void *item) {
  static char string[100];
  snprintf(string, sizeof string, "%d", *(int *)item);
  return string;
}

const char *list_list_to_string(const void *item) {
  return list_to_string(item, int_to_string);
}

int main(void) {
  list_init();
  {
    unsigned int i;

    int tab1[] = {0, 1, 2, 3, 4};
    List *list1 = NULL;
    for (i = 0; i < sizeof(tab1) / sizeof(*tab1); i++) {
      list1 = list_append(list1, tab1 + i);
    }

    int tab2[] = {5, 6, 7, 8, 9};
    List *list2 = NULL;
    for (i = 0; i < sizeof(tab2) / sizeof(*tab2); i++) {
      list2 = list_append(list2, tab2 + i);
    }

    List *list = NULL;
    list = list_append(list, list1);
    list = list_append(list, list2);

    assert(strcmp(list_to_string(list, list_list_to_string),
                  "[[0,1,2,3,4],[5,6,7,8,9]]") == 0);
```

```
    list_destroy(list1);
    list_destroy(list2);
    list_destroy(list);
  }
  list_finish();
  return EXIT_SUCCESS;
}
```

ne devra générer aucune erreur.

2.2 Itérations sur les listes génériques

Les listes représentant une collection de données, il est naturel de pouvoir utiliser le *patron* de programmation *Itérateur*.

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

#include "./list.h"

int main(void) {
  int a[] = {1, 2, 3, 4, 5, 6};
  int i;

  List *list = NULL;
  ListIterator *iterator;
  for (i = 0; i < sizeof(a) / sizeof(*a); i++) {
    list = list_append(list, a + i);
  }

  for (
    iterator = list_iterator_create(list);
    list_iterator_valid(iterator);
    iterator = list_iterator_next(iterator)
  ) {
    printf("%d\n", *(int *)list_iterator_get_data(iterator));
  }

  list_iterator_destroy(iterator);
  list_destroy(list);
  return EXIT_SUCCESS;
}
```

Ce code permet d'afficher les lignes suivantes :

```
1
2
3
```

4
5
6

Exercice 30 *(Définition du type `ListIterator`)*
Définissez un nouveau type `ListIterator` dans le fichier `list.h` permettant d'itérer sur une liste.

Exercice 31 *(Allocation et désallocation des pointeurs vers `ListIterator`)*
Écrivez deux fonctions

```
extern ListIterator *list_iterator_create(const List * list);
extern void list_iterator_destroy(ListIterator * iterator);
```

permettant de créer un itérateur et de libérer sa mémoire.

Exercice 32 *(Implémentation du fonctionnement de `ListIterator`)*
Écrivez trois fonctions

```
extern ListIterator *list_iterator_rewind(ListIterator * iterator);
extern ListIterator *list_iterator_next(ListIterator * iterator);
extern bool list_iterator_valid(const ListIterator * iterator);
```

permettant d'itérer à partir d'un itérateur.

- `list_iterator_next` permet de faire avancer l'itérateur à la prochaine donnée ;
- `list_iterator_rewind` permet de réinitialiser l'itérateur sur la première donnée ;
- `list_iterator_valid` permet de savoir s'il reste encore des éléments sur lesquels itérer dans l'itérateur ;

Exercice 33 *(Accès à la donnée de `ListIterator`)*

1. Écrivez une fonction

   ```
   extern void *list_iterator_get_data(const ListIterator * iterator);
   ```

   permettant d'accéder aux données de l'élément courant d'un itérateur.
2. Écrivez un programme de test `test-iterate` permettant de valider l'ensemble des fonctions écrites sur les itérateurs de listes.

Exercice 34 *(Retour sur l'allocation des listes)*
Modifiez les fonctions libérant et allouant la mémoire de manière à ce qu'elles ne la libèrent pas. L'appel à la fonction `free` sera remplacé par l'appel à une fonction `_list_keep` permettant de garder une liste des nœuds déjà alloués. Cette liste des nœuds déjà alloués sera stockée dans une variable statique `available`. Ces noœuds pourront être réutilisé lors de l'appel à la fonction `malloc` qui sera remplacée par un appel à une fonction `_list_create`. Cette fonction `_list_create` regardera tout d'abord si un élément est disponible dans la liste `available` avant d'allouer un nouvel élément si il n'y en a plus de disponible.
Faites en sorte que la liste `available` ne puissent pas dépasser un certain seuil que l'utilisateur de la bibliothèque pourra préciser au moyen d'une variable globale `list_available_max`.

Historique des modifications

2011-2012_1 *Lundi 13 février 2012*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Création initiale du document

2012-2013_1 *Lundi 4 mars 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Création initiale du document

2012-2013_2 *Samedi 16 mars 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout des listes

2012-2013_3 *Lundi 18 mars 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Corrections typographiques et Ajout de précisions.

2012-2013_4 *Mercredi 20 mars 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'un exercice `list_index`.

2012-2013_5 *Lundi 25 mars 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout des tables de dispersion.

2012-2013_6 *Mardi 9 avril 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout des arbres AVL cousus.

2013-2014_1 *Lundi 20 janvier 2014*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Réutilisation de la mémoire pour les lites.

2013-2014_2 *Dimanche 9 mars 2014*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout des schémas pour les tables de dispersions et les arbres AVL cousus.

2013-2014_3 *Mercredi 12 mars 2014*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction d'une coquille orthographique ;

— Renommage de fonctions.

2013-2014_4 *Mercredi 2 avril 2014*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout de mots-clés `const` dans certaines fonctions ;

— Amélioration du schéma des tables de dispersion.

2014-2015_1 *Mardi 24 mars 2015*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année.

2014-2015_2 *Jeudi 2 avril 2015*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Passage de `const void *` pour les données de listes.

2015-2016_1 *Mercredi 3 février 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année ;
— Passage au format *pandoc*.

2015-2016_2 *Lundi 29 février 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Renommage des programmes de test sur les tableaux dynamiques.

2015-2016_3 *Mercredi 2 mars 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Suppression de la contrainte `max >= 1` pour les tableaux dynamiques ;
— Ajout d'une précision sur le ratio pour les tableaux dynamiques.

2016-2017_1 *Lundi 13 mars 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année.

2016-2017_2 *Lundi 13 mars 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction grammaticale.

2016-2017_3 *Mercredi 15 mars 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Proposition d'un nouveau mode de calcul de la longueur maximum d'un tableau dynamique.

2016-2017_4 *Mardi 4 avril 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Proposition de 3 modes de tri pour les listes : tri à pivot, tri à fusion, tri à bulles.

2016-2017_5 *Dimanche 2 avril 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'exercice pour les tableaux dynamiques : conversion d'un tableau en chaines de caractères ;
— Ajout d'exercice pour les listes : conversion d'une liste en chaines de caractères ;
— Ajout d'exercice pour les listes : inversion d'une liste.

2017-2018_1 *Lundi 5 février 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année ;

— Renommage des fonctions d'initialisation et de finalisation.

2017-2018_2 *Vendredi 16 mars 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'un mot-clé **`extern`** dans la déclaration de la fonction `array_to_string` (voir exercice 7 (conversion en chaîne de caractères des collections dynamiques)) ;
— Retrait du qualificateur `const` sur le paramètre `ratio` (voir exercice 3 (fonctions de création et de destruction sur les collections dynamiques)) pour permettre sa modification dans la fonction.

2018-2019_1 *Jeudi 15 novembre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année ;
— Changement du formattage.

2018-2019_2 *Jeudi 15 novembre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'un mot clef **`extern`**.

2019-2020_1 *Vendredi 18 octobre 2019*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Utilisation du template https://github.com/Wandmalfarbe/pandoc-latex-template.

2020-2021_1 *Lundi 12 octobre 2020*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'une page de garde en couleur ;
— Changement d'année.

2022-2023_1 *Lundi 2 janvier 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Adaptation à la nouvelle offre de formation.

2022-2023_2 *Lundi 16 janvier 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Renommage de *tableau dynamique* en *collection dynamique*. Le mot *tableau* reste réservé pour les tableaux déclaré sous la forme `type tab[n]`.

2022-2023_3 *Vendredi 20 janvier 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement de la page de titre.

2022-2023_4 *Lundi 23 janvier 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Renommage des fonctions `*_vals` et `*_val` en `*_values` et `*_value`.

2022-2023_5 *Mercredi 25 janvier 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Division de l'exercice d'insertion et de suppression de valeurs dans les collections dynamiques en deux exercices (voir exercice 5 (insertion de valeurs dans une collection dynamique) et exercice 6 (suppression de valeurs dans une collection dynamique)).
— Précision sur la place du type `ArrayIterator` dans l'exercice 8 (définition du type `ArrayIterator`).
— Suppression de la question sur le test de l'itérateur des collections dynamiques car le test évolue avec les exercices.

2022-2023_6 *Jeudi 26 janvier 2023*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Typo dans les noms de fonctions `*_values`.