Mémoire

MI074 - 3

Plan

- Gestion de la mémoire dynamique
- Gestion des lignes chainées

Gestion mémoire

Un programme peut créer des structures dans 3 types d'espace:

1. Dans des variables globales dont les adresses sont connues dès la phase de compilation-édition des liens.
2. Dans des variables locales dans la pile. C'est un espace relatif au pointeur de pile qui n'est utilisable que pendant la durée de vie de la fonction qui déclare l'espace.
3. Dans le tas (heap). C'est un espace global géré dynamiquement dans lequel on peut réserver des zones contiguës par malloc et que l'on doit explicitement libérer par free.

Fonctions de la mémoire dynamique

void *malloc(size_t size)
renvoie un pointeur vers un bloc de mémoire virtuelle de taille au moins égale à size (ajusté selon alignement ; en général frontière de double mot : 8 octets). Si erreur (par ex. pas assez de place disponible), malloc() renvoie NULL et affecte une valeur au code d! erreur errno. Attention : la mémoire n!est pas initialisée. La primitive calloc() fonctionne comme malloc() mais initialise le bloc alloué à 0.

void free(void *ptr)
doit être appelé avec une valeur de ptr rendue par un appel de malloc(). Son effet est de libérer le bloc alloué lors de cet appel. Attention : pour une autre valeur de ptr, l!effet de free() est indéterminé.

Solution naïve

Si la mémoire allouée n'est jamais libérée alors on peut utiliser une solution naïve mais efficace utilisant un pointeur de limite.

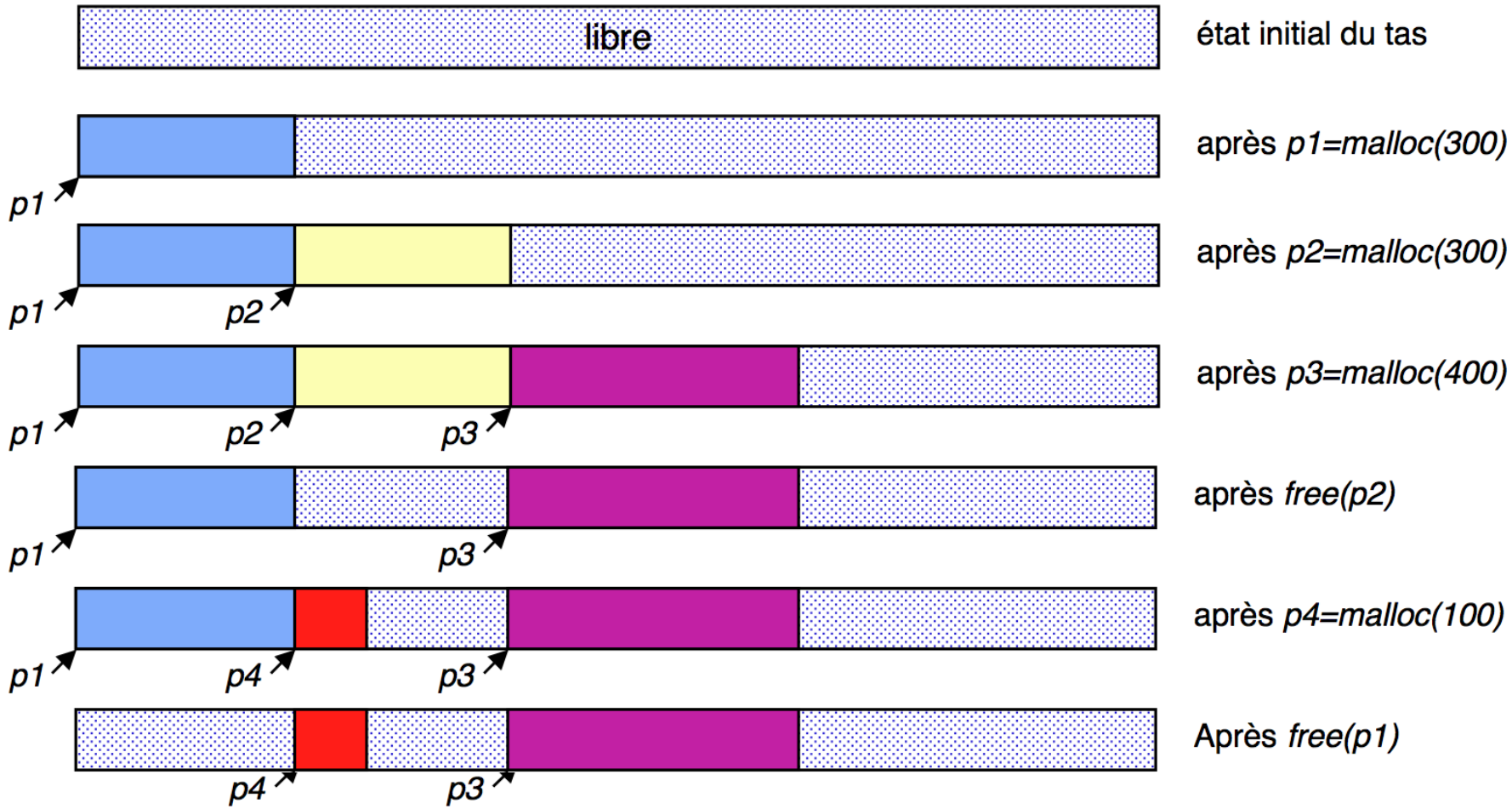
On suppose disposer :

- d'un espace contiguë de grande taille.
- d'un pointeur TOP initialisé sur la 1ère case de cet espace.

A chaque nouveau malloc :

- on se contente de rendre la valeur du pointeur TOP.
- et on repousse TOP de la taille de la zone demandée.

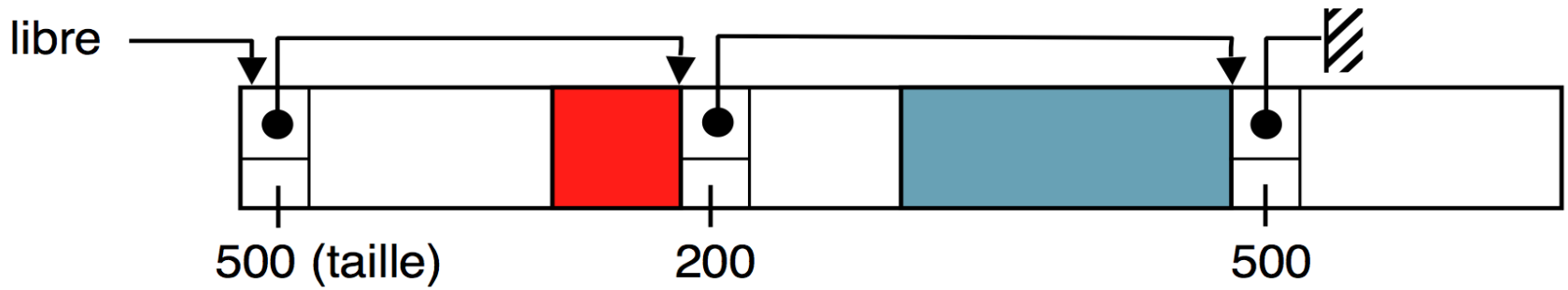
Gestion malloc-free



- Pour libérer et allouer dans les espaces libérés (p4) il faut se souvenir de la taille des zones allouées.
- On voit apparaitre un phénomène de fragmentation qui entraine de perte de mémoire

Structuration et recherche dans le tas

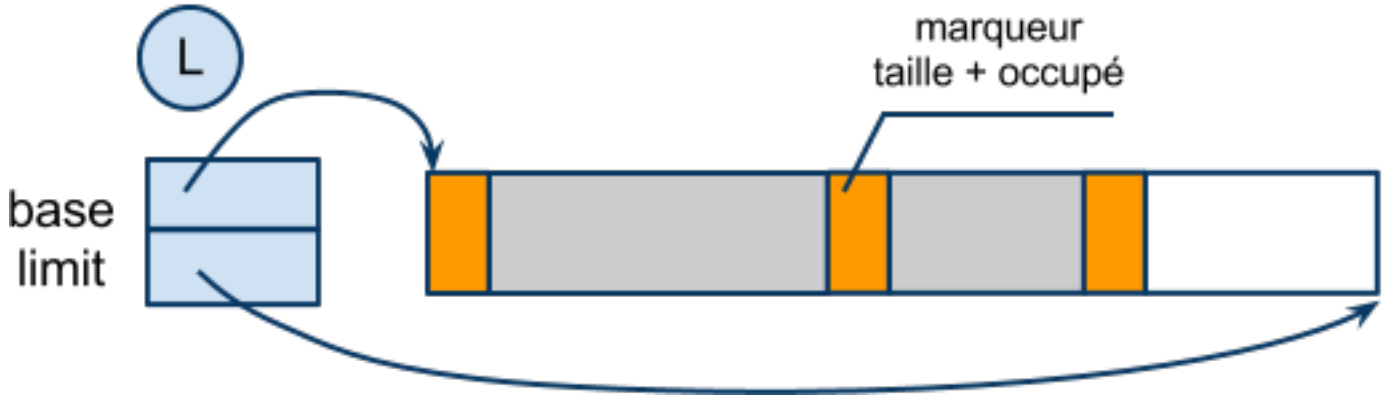
Représentation des zones libres : liste chaînée, pointeurs dans zones libres



Pour satisfaire une demande, on parcourt la liste libre, et on peut prendre :

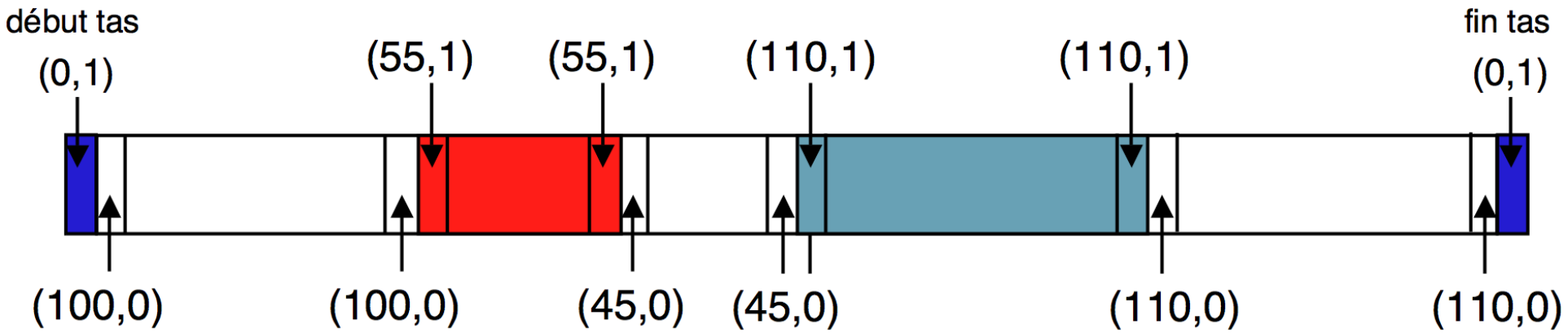
1. La première zone libre assez grande pour satisfaire la demande (**first fit**).
   Avantage : rapidité.
   Inconvénient : risque de mauvaise utilisation de l'espace libre (perte de grands segments), on parle de fragmentation externe.
   Optimisation (**next fit**) : la recherche commence à partir de la position courante du pointeur de parcours (et non à la première zone libre).
   Avantage : évite l'accumulation de petites zones libres en début de liste.
2. La zone dont la taille est plus proche de la demande (**best fit**).
   Avantage : meilleur ajustement.
   Inconvénients : plus lent ; risque de créer de nombreuses zones de faible taille (émiettement), on parle de fragmentation interne.

Implémentation des pointeurs de zones



- Les pointeurs de zones sont remplaçables par des marqueurs qui codent sur un entier : la taille de la zone et son état d'occupation.
- Connaissant le pointeur de début, on peut parcourir les zones à la recherche d'espace libre.
- Lors des libérations on peut fusionner les zones contiguës (dans les adresses croissantes)

Optimisations



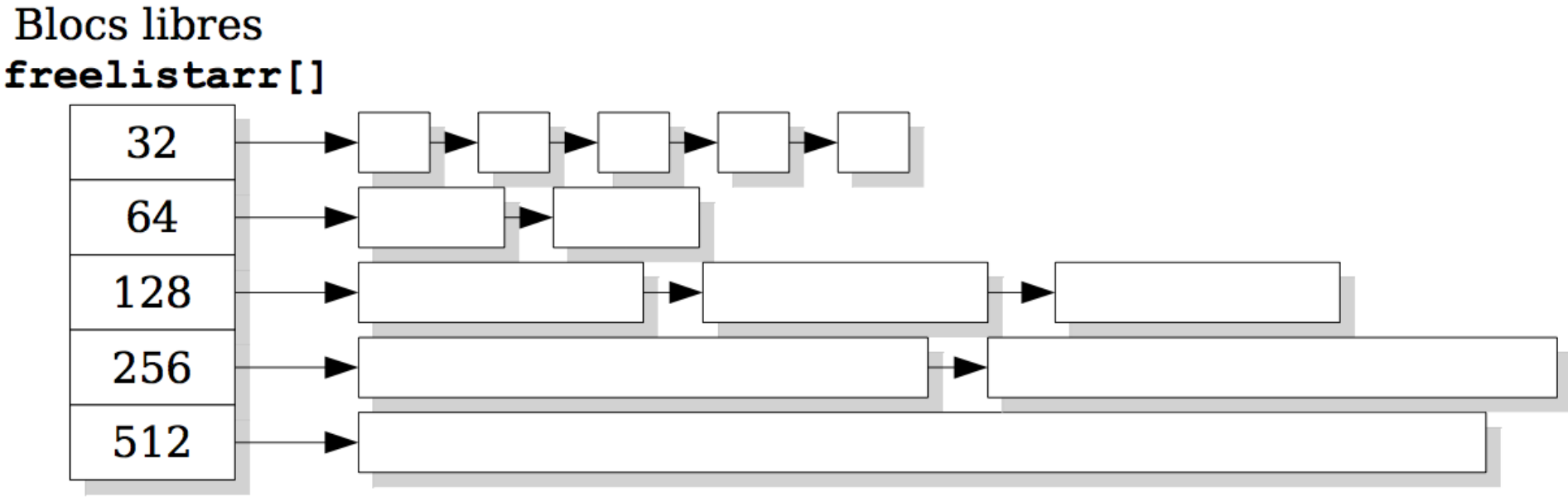
Fusion des zones libres:

- En utilisant deux marqueurs, on peut fusionner dans les deux sens. Il suffit que deux marqueurs consécutifs soient libres.
- Les marqueurs de début et de fin de tas permettent de simplifier l'algorithme de fusion.

Comportement des caches :

- Les zones allouée sont alignées sur des lignes de caches. Cela doit réduire le taux de miss et évite que deux structures allouées partagent la même ligne.

Optimisations



- Pour réduire la fragmentation externe, on peut créer plusieurs tas où chaque tas gère un intervalle de taille.
- Lorsque que le tas est très utilisé avec un grand nombre d'allocation-libération. on peut créer des pools de buffers de taille standardisée. L'allocation initiale est faite dans le tas, la libération attache la zone libérée à une liste de buffer libres (1 liste par taille).

Les listes chainées

La plupart des structures du données du système utilise des listes chainées. Il faut disposer d'une API permettant de :

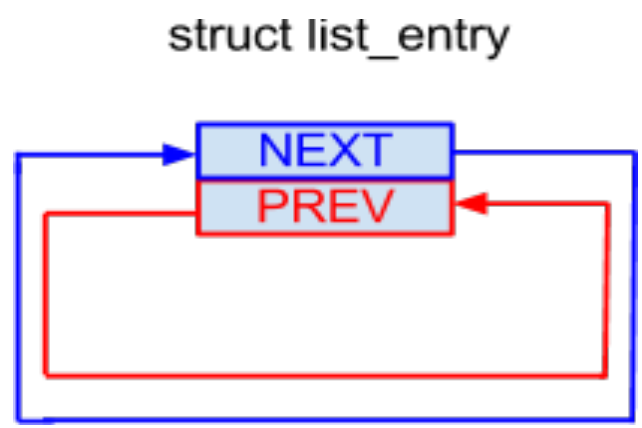
- créer
- détruire
- insérer
- parcourir

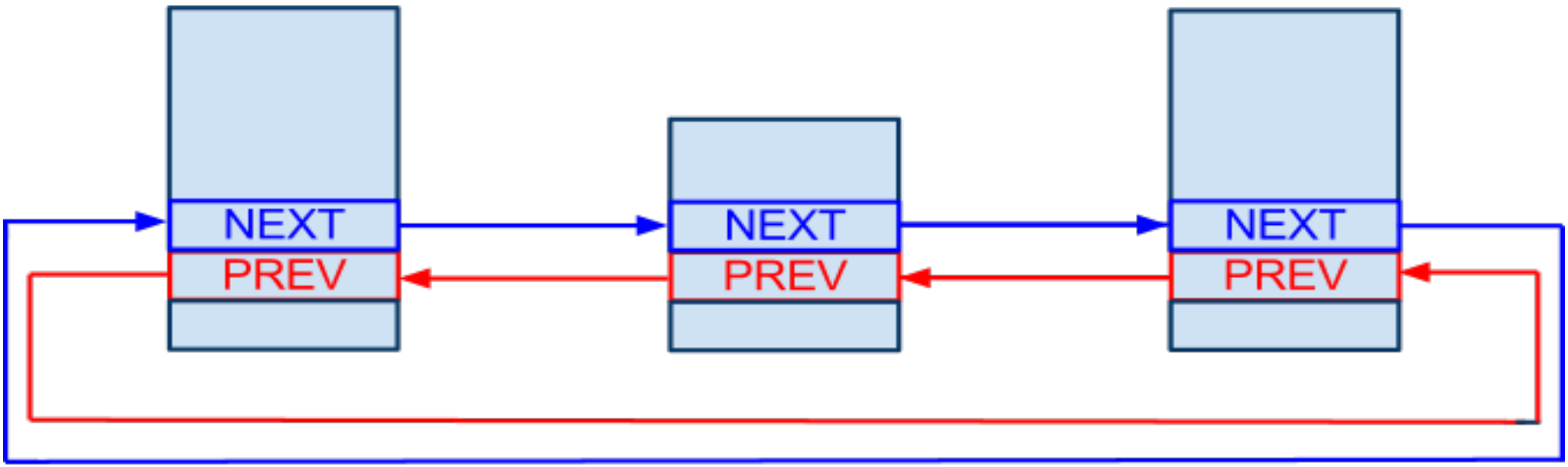
pour les manipuler de façon homogène et lisible sans ajouter de surcoût en mémoire et en temps vis à vis d'une solution traditionnelle.

Gestion des listes chainées

Le système utilise des listes chainées partout. le noyau exige un mécanisme générique et efficace.

- Déclaration
- Itérateur
- Fonctions d'accès
- Exemple





Gestion des listes chainées : exemple

```c
#include "list.h"
#include <stdio.h>

// Definition du type famille
struct famille_s
{
  char *nom;
  struct list_entry root;
};

// Definition du type personne
struct personne_s
{
  char *prenom;
  struct list_entry list;
};
int main()
{
  struct famille_s dupont;
  struct personne_s jean_claude;
  struct personne_s monique;

  struct personne_s *ptr;
  struct list_entry *iter;

  dupont.nom = "DUPONT";
  jean_claude.prenom = "Jean-Claude";
  monique.prenom = "Monique";

  // Initialisation de la racine de list
  list_root_init(&dupont.root);

  // Ajout d'un personne
  list_add(&dupont.root, &monique.list);

  // Ajout d'une autre personne
  list_add(&dupont.root, &jean_claude.list);

  printf("La famille %s :\n", dupont.nom);
  list_foreach(&dupont.root,iter)
  {
    ptr = list_element(iter, struct personne_s, list);
    printf("%s, ", ptr->prenom);
  }
  printf("\b\b \n");

  return 0;
}
```

Gestion des listes chainées API

```
LIST_ROOT_INITIALIZER(name)
list_root_init(struct list_entry *root)
list_entry_init(struct list_entry *entry)

list_foreach_forward(root_ptr, iter)
list_foreach_backward(root_ptr, iter)
list_foreach(root_ptr, iter)

list_first(root_ptr, type, member)
list_last(root_ptr, type, member)
list_empty(root)

list_element(list_ptr, type, member)

list_add(struct list_entry *root, struct list_entry *entry)
list_add_first(root, entry)
list_add_next(root, entry)
list_add_last(root, entry)
list_add_pred(root, entry)

list_unlink(struct list_entry *entry)
list_replace(struct list_entry *current, struct list_entry *new)

list_container(struct list_entry *entry, container_type,
member_name)
```