I3 – Algorithmique et programmation
Introduction à la programmation en langage C
Cours n°6
Listes chaînées et débuggage avec gdb

Camille Coti
camille.coti@iutv.univ-paris13.fr

IUT de Villetaneuse, département R&T

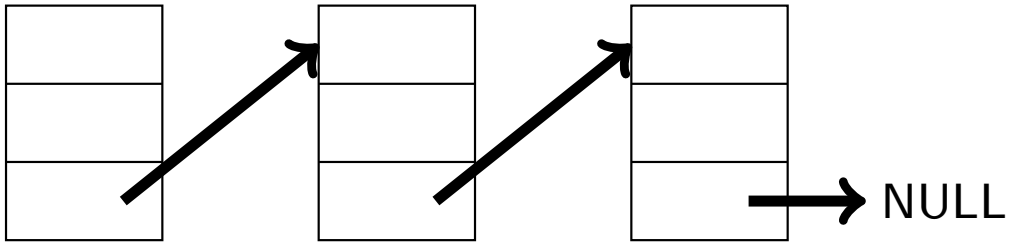
2011 – 2012

Définition

Liste chaînée

Une liste chaînée est une **structure de données** permettant de stocker des données de façon **ordonnée** et **dynamique**. Chaque maillon de la chaîne est une structure de données contenant

- Les données stockées
- Un pointeur vers l'élément suivant



Le pointeur du dernier élément ne pointe vers rien : on lui assigne la valeur `NULL`.

Exemple : structure de données

Déclaration en C d'une structure de données contenant obligatoirement :

- Les données à stocker dans chaque élément
- Un pointeur vers l'élément suivant

Exemple :

```
1 struct element {
2   int indice;
3   float valeur;
4   struct element* suivant;
5 };
```

Ici :

- La structure s'appelle `element`
- Les données à stocker sont un entier et un flottant
- On a un pointeur sur une variable de type `struct element` : on peut pointer vers un `element`

Éléments d'une liste

Le début de la liste est un pointeur vers le premier élément

struct element* liste; → NULL

On accède à l'élément suivant en suivant le pointeur d'un élément donné


```
1 elem = elem->suivant;
```

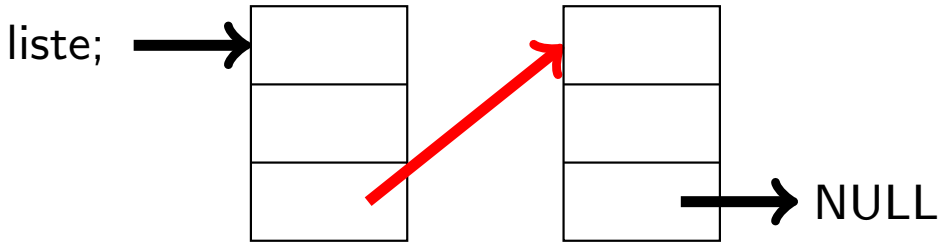
Le dernier élément n'a pas de successeur : son pointeur est à NULL

Éléments d'une liste

Le début de la liste est un pointeur vers le premier élément

struct element* liste; → NULL

On accède à l'élément suivant en suivant le pointeur d'un élément donné




```
1 elem = elem->suivant;
```

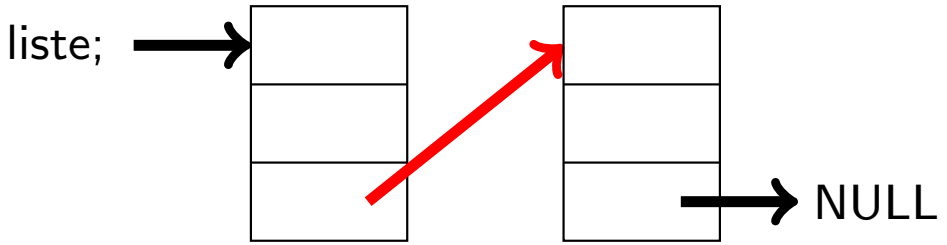
Le dernier élément n'a pas de successeur : son pointeur est à NULL

Éléments d'une liste

Le début de la liste est un pointeur vers le premier élément

struct element* liste; → NULL

On accède à l'élément suivant en suivant le pointeur d'un élément donné




```
1 elem = elem->suivant;
```

Le dernier élément n'a pas de successeur : son pointeur est à NULL

Opérations sur les éléments d'une liste

Création d'une liste

- Déclaration d'un **pointeur vers un élément de la liste**

Ajout d'un élément au début d'une liste

- Le pointeur du nouvel élément pointe vers le reste de la liste
- Le pointeur de début de liste pointe vers le nouvel élément

Ajout d'un élément en fin d'une liste

- Le pointeur du dernier élément pointe vers le nouvel élément
- Le pointeur du nouvel élément est mis à `NULL`

Ajout d'un élément dans une liste

- Le pointeur de l'élément précédent pointe vers le nouvel élément
- Le pointeur du nouvel élément pointe vers l'élément suivant

Suppression d'un élément d'une liste

- Le pointeur de l'élément précédent pointe vers l'élément suivant

Création d'une liste chaînée

On parcourt une liste chaînée en utilisant le pointeur de la structure de données de ses éléments

- Le pointeur pointe sur **l'élément suivant**
- On l'utilise pour passer d'un élément à un autre **dans un seul sens**
- Fin de la liste : quand le pointeur vaut **NULL**

```
1  struct element {
2    int indice;
3    float valeur;
4    struct element* suivant;
5  };
6
7  /* On a une liste d'éléments */
8  struct element* elem_actuel;
9
10 elem_actuel = liste;
11 while( elem_actuel->suivant != NULL ) {
12   elem_actuel = elem_actuel->suivant;
13   printf( "elem actuel: %f\n", valeur );
14 }
```

Création d'une liste chaînée

- Déclaration d'un **pointeur vers un élément de la liste**

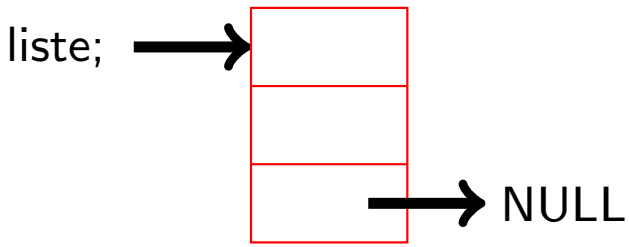
```
1  struct element {
2    int indice;
3    float valeur;
4    struct element* suivant;
5  };
6
7  struct element* liste = NULL;
```

struct element* liste; → NULL

Ajout d'un élément au début d'une liste chaînée

- Le pointeur du nouvel élément pointe vers le reste de la liste
- Le pointeur de début de liste pointe vers le nouvel élément

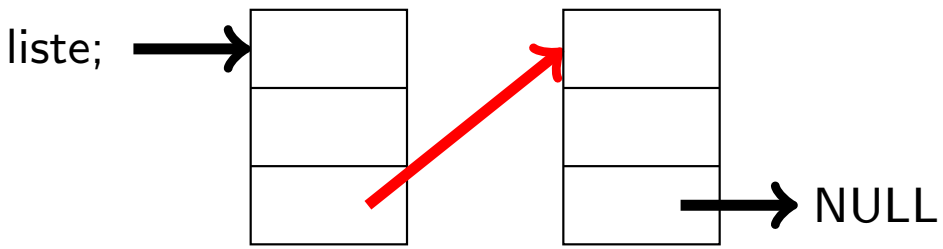
```
1  struct element {
2    int indice;
3    float valeur;
4    struct element* suivant;
5  };
6
7  struct element* liste;
8  struct element* elem;
9
10 /* Creation de l'element */
11 elem = (struct element*)malloc(sizeof(struct element));
12
13 /* Ajout de l'element */
14 elem->suivant = NULL;
15 liste = elem;
```



Ajout d'un élément à la fin d'une liste chaînée

- Le pointeur du dernier élément pointe vers le nouvel élément
- Le pointeur du nouvel élément est mis à `NULL`

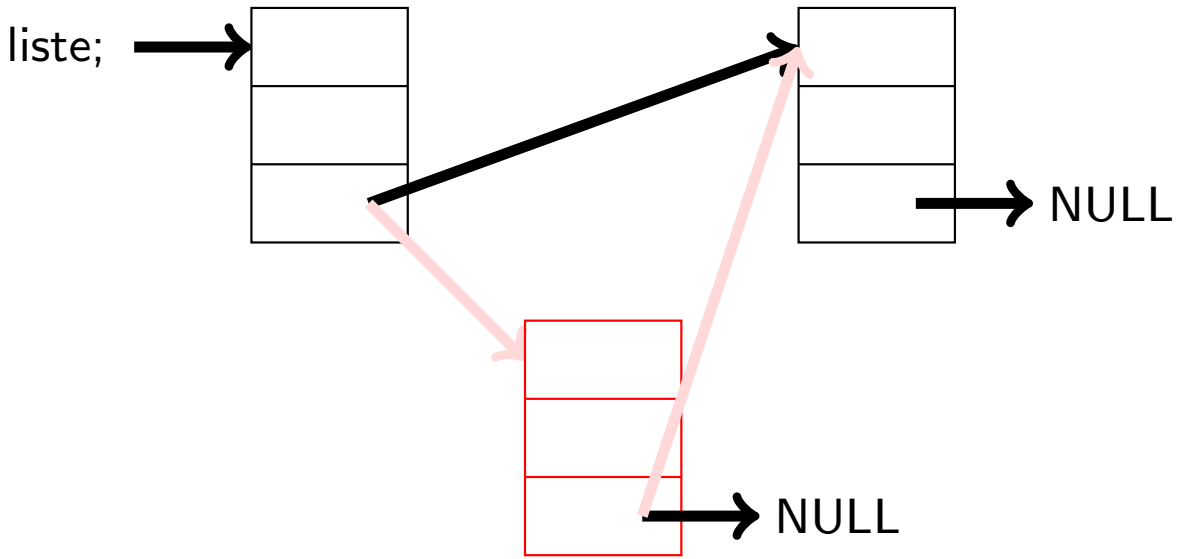
```
1  /* On a une liste dont le dernier element est pointé par elem */
2  struct element* nouv_elem:
3
4  /* Creation de l'element */
5  nouv_elem = (struct element*)malloc(sizeof(struct element));
6
7  /* Ajout de l'element */
8  elem->suivant = nouv_elem;
9  nouv_elem->suivant = NULL;
```



Ajout d'un élément dans une liste chaînée

- Le pointeur de l'élément précédent pointe vers le nouvel élément
- Le pointeur du nouvel élément pointe vers l'élément suivant

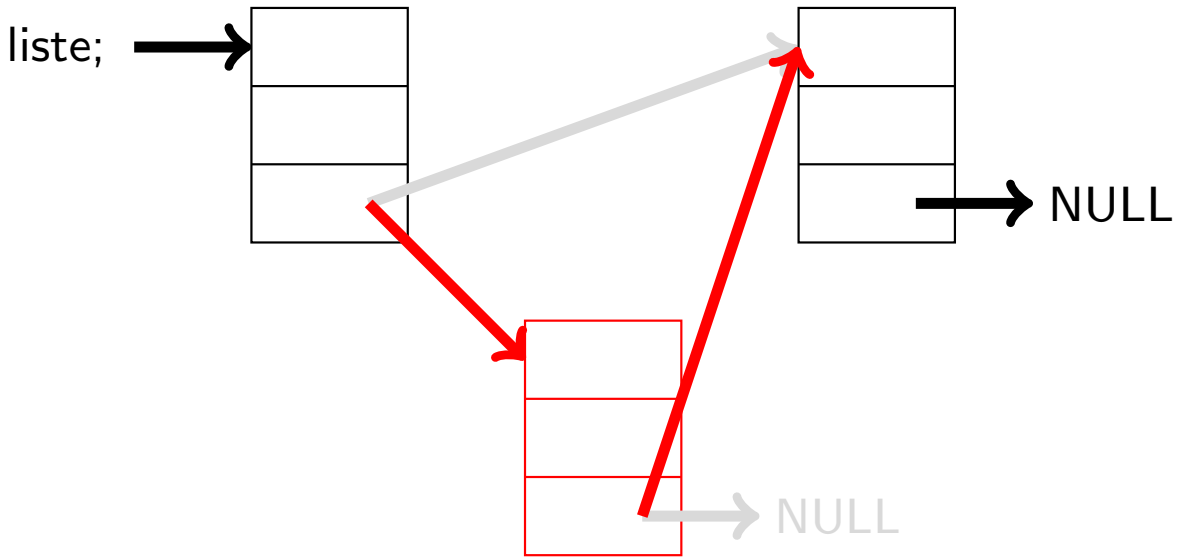
```
1  /* On a une liste et on est sur un element elem de la liste */
2  struct element* nouv_elem:
3  /* Creation de l'element */
4  nouv_elem = (struct element*)malloc(sizeof(struct element));
5  /* L'element suivant est pointé par le pointeur du nouvel element */
6  nouv_elem->suivant = elem->suivant;
7  /* Ajout du nouvel element derrière l'element precedent elem */
8  elem->suivant =nouv_elem;
```



Ajout d'un élément dans une liste chaînée

- Le pointeur de l'élément précédent pointe vers le nouvel élément
- Le pointeur du nouvel élément pointe vers l'élément suivant

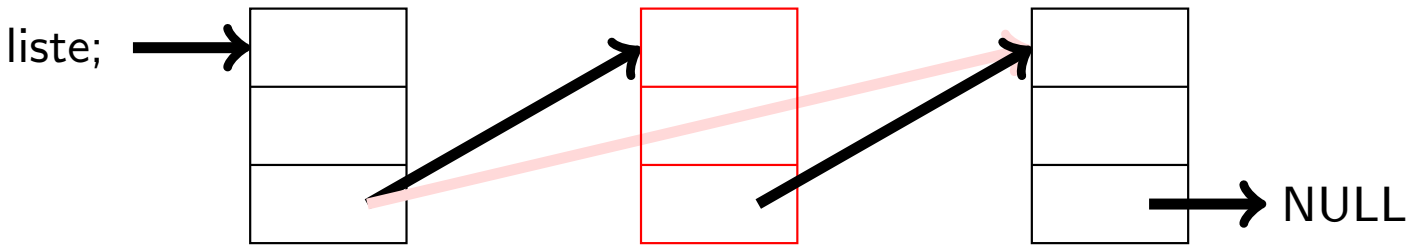
```
1  /* On a une liste et on est sur un element elem de la liste */
2  struct element* nouv_elem:
3  /* Creation de l'element */
4  nouv_elem = (struct element*)malloc(sizeof(struct element));
5  /* L'element suivant est pointé par le pointeur du nouvel element */
6  nouv_elem->suivant = elem->suivant;
7  /* Ajout du nouvel element derrière l'element precedent elem */
8  elem->suivant =nouv_elem;
```



Suppression d'un élément d'une liste chaînée

- Le pointeur de l'élément précédent pointe vers l'élément suivant

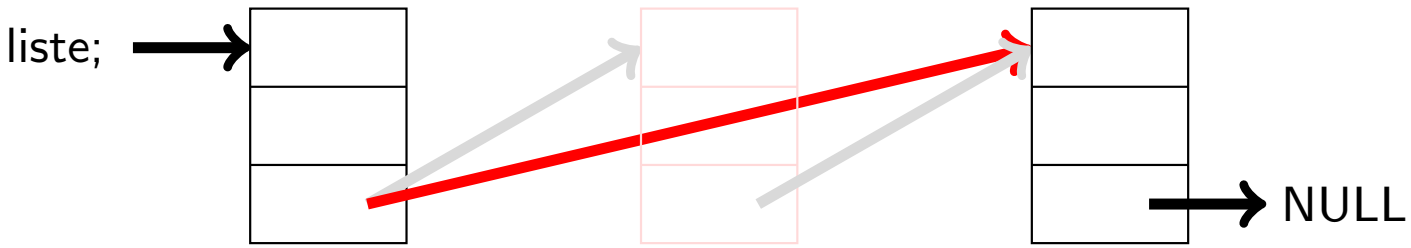
```
1  /* On a une liste et on est sur un element elem de la liste */
2  struct element* tmp;
3  /* On garde le pointeur vers l'element suivant */
4  tmp = elem->suivant;
5  /* On fait pointer l'element precedent vers l'element suivant */
6  elem->suivant = tmp->suivant;
7  /* Destruction de l'element */
8  free( tmp );
```



Suppression d'un élément d'une liste chaînée

- Le pointeur de l'élément précédent pointe vers l'élément suivant

```
1  /* On a une liste et on est sur un element elem de la liste */
2  struct element* tmp;
3  /* On garde le pointeur vers l'element suivant */
4  tmp = elem->suivant;
5  /* On fait pointer l'element precedent vers l'element suivant */
6  elem->suivant = tmp->suivant;
7  /* Destruction de l'element */
8  free( tmp );
```



Propriétés des éléments d'une liste chaînée : ordre

Ordre dans une liste chaînée

Les éléments d'une liste chaînée peuvent être **ordonnés** selon un ordre choisi par le programmeur.

L'ordre est défini au moment de l'insertion d'un élément dans la liste.

```
1  /* On a une liste et on veut ajouter un element dans la liste */
2  struct element* tmp;
3  struct element* prec;
4
5  while( ( nouv_elem->valeur < tmp->valeur )
6         && ( tmp->suivant != NULL ) ) {
7    prec = tmp;
8    tmp = tmp->suivant;
9  }
10 if( tmp->suivant == NULL ) {
11   tmp->suivant = nouv_elem;
12   nouv_elem->suivant = NULL;
13 } else {
14   prec->suivant = nouv_elem;
15   nouv_elem->suivant = tmp;
16 }
```

Propriétés des éléments d'une liste chaînée : taille

Taille d'une liste chaînée

La taille d'une liste chaînée est **dynamique** :

- Elle augmente lorsqu'on insère un nouvel élément
- Elle diminue lorsqu'on retire un élément

C'est une différence très importante avec les tableaux qui, eux, ont une taille fixe !

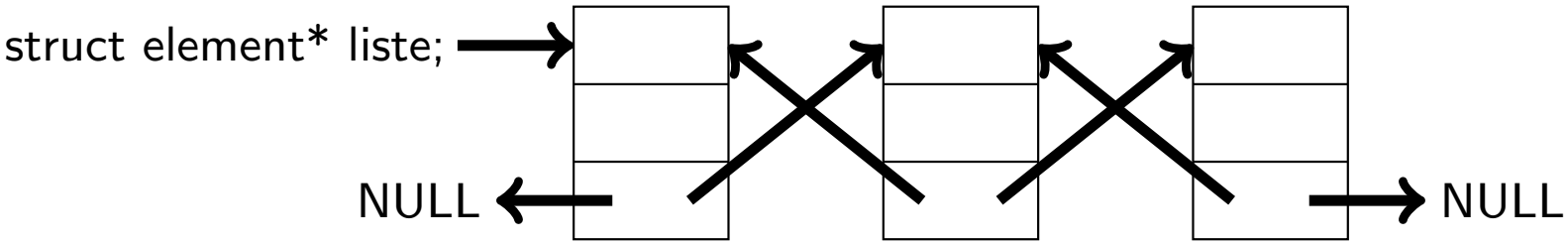
```
1  /* Fonction retournant le nombre d'elements dans une liste */
2  int tailleListe( struct element* lst ) {
3    struct element* elem_courant;
4
5    int taille = 0;
6    elem_courant = lst;
7
8    while( elem_courant != NULL ) {
9      taille++;
10     elem_courant = elem_courant->suivant;
11   }
12
13   return taille;
14 }
```

Listes doublement chaînées

Liste doublement chaînée : définition

Une liste doublement chaînée est une **structure de données** permettant de stocker des données de façon **ordonnée** et **dynamique**. Chaque maillon de la chaîne est une structure de données contenant

- Les données stockées
- Un pointeur vers l'élément suivant : le **pointeur avant**
- Un pointeur vers l'élément précédent : le **pointeur arrière**



- Le pointeur avant du dernier élément ne pointe vers rien : on lui assigne la valeur `NULL`.
- Le pointeur arrière du premier élément ne pointe lui non plus vers rien. On lui assigne aussi la valeur `NULL`.

Listes doublement chaînées : exemple

```
1  struct element {
2    int indice;
3    float valeur;
4    struct element* avant;
5    struct element* arriere;
6  };
```

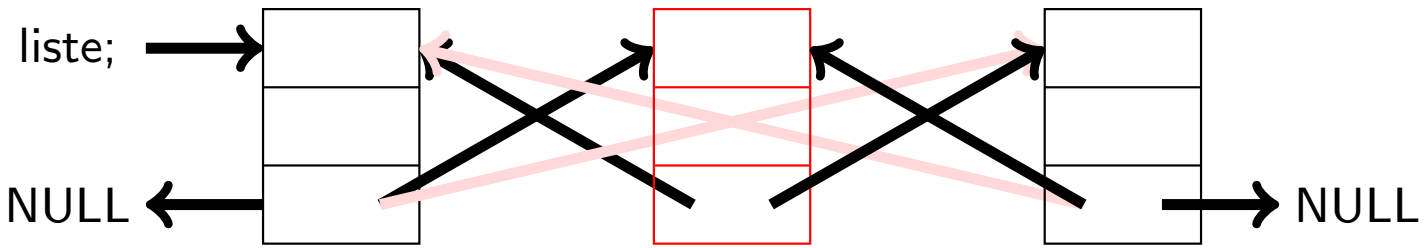
Différences avec les listes simplement chaînées :

- On peut **revenir en arrière** dans le parcours d'une liste doublement chaînée
- Attention, les opérations effectuées doivent tenir compte des **deux pointeurs** !

Suppression d'un élément d'une liste doublement chaînée

- Le pointeur avant de l'élément précédent pointe vers l'élément suivant
- Le pointeur arrière de l'élément suivant pointe vers l'élément précédent

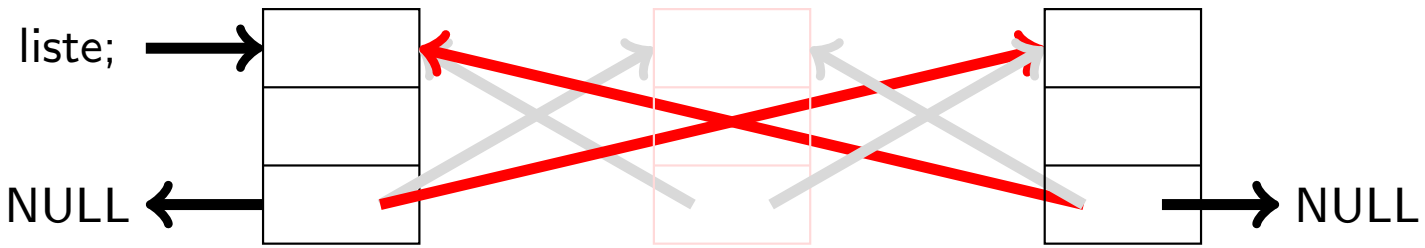
```
1  /* On a une liste et on est sur un element elem de la liste */
2  struct element* tmp;
3
4  /* On garde le pointeur vers l'element suivant */
5  tmp = elem->avant;
6
7  /* On fait pointer l'element precedent vers l'element suivant */
8  elem->avant = tmp->avant;
9  /* On fait pointer l'element suivant vers l'element precedent */
10 tmp = elem->avant;
11 tmp->arriere = elem;
12
13 /* Destruction de l'element */
14 free( tmp );
```



Suppression d'un élément d'une liste doublement chaînée

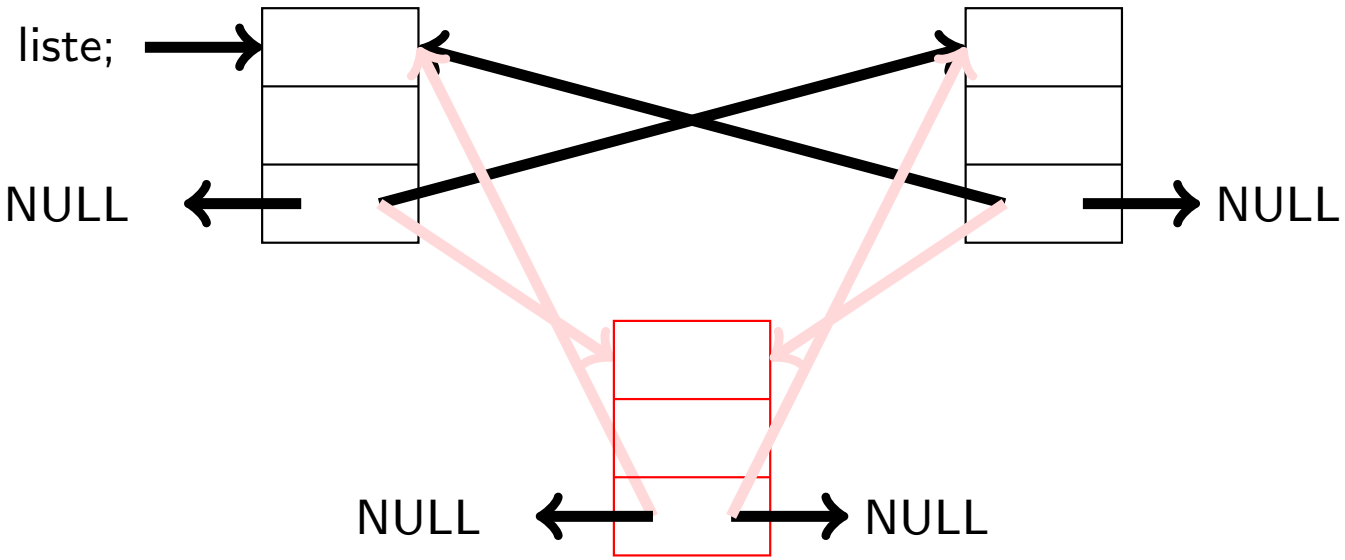
- Le pointeur avant de l'élément précédent pointe vers l'élément suivant
- Le pointeur arrière de l'élément suivant pointe vers l'élément précédent

```
1  /* On a une liste et on est sur un element elem de la liste */
2  struct element* tmp;
3
4  /* On garde le pointeur vers l'element suivant */
5  tmp = elem->avant;
6
7  /* On fait pointer l'element precedent vers l'element suivant */
8  elem->avant = tmp->avant;
9  /* On fait pointer l'element suivant vers l'element precedent */
10 tmp = elem->avant;
11 tmp->arriere = elem;
12
13 /* Destruction de l'element */
14 free( tmp );
```



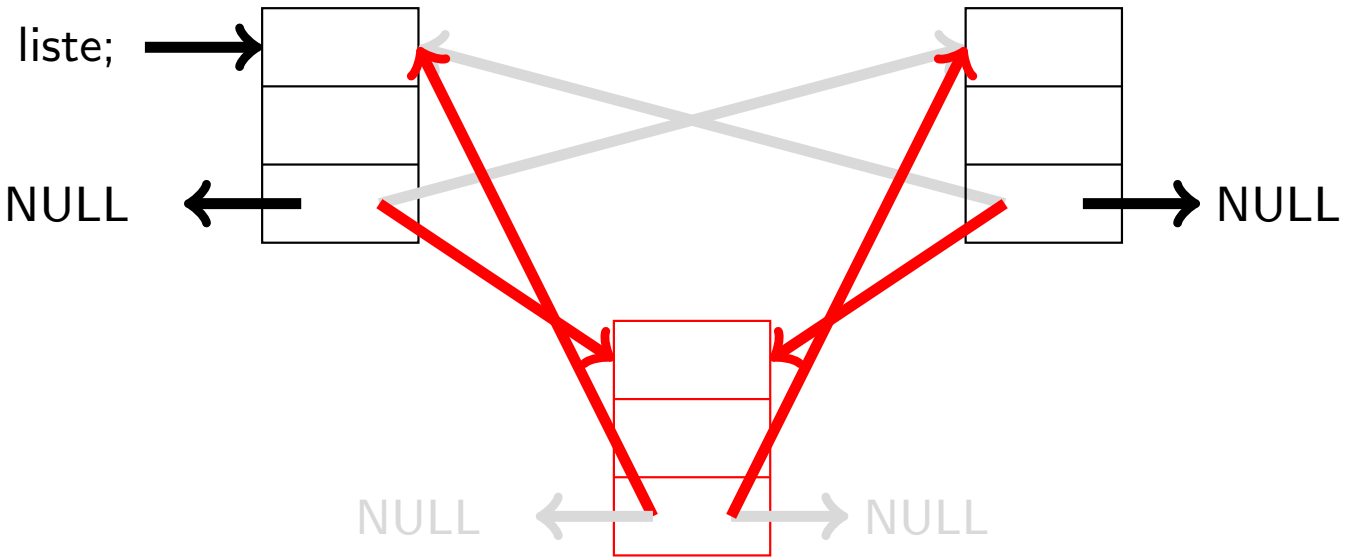
Ajout d'un élément dans liste doublement chaînée

- Le pointeur avant de l'élément précédent pointe vers le nouvel élément
- Le pointeur arrière du nouvel élément pointe vers l'élément précédent
- Le pointeur avant du nouvel élément pointe vers l'élément suivant
- Le pointeur arrière de l'élément suivant pointe vers le nouvel élément



Ajout d'un élément dans liste doublement chaînée

- Le pointeur avant de l'élément précédent pointe vers le nouvel élément
- Le pointeur arrière du nouvel élément pointe vers l'élément précédent
- Le pointeur avant du nouvel élément pointe vers l'élément suivant
- Le pointeur arrière de l'élément suivant pointe vers le nouvel élément



GNU Debugger

GNU Debugger (gdb) est un outil de mise au point et de débuggage de programmes

- Il permet d'**observer** le comportement d'un programme
- **Pendant l'exécution** ou **post-mortem** (examen du core dump généré par le système pour un processus ayant planté)
- On peut **accéder aux valeurs** des variables
- En **modifier** la valeur
- Exécuter le programme **instruction par instruction**
- Définir des **points d'arrêt** dans le programme

Utilisation :

- Compilation du programme avec l'option `-g`
  - `gcc -g -o toto toto.c`
  - Génère des informations sur l'exécutable, liens avec le code source
- Exécution du programme dans gdb
  - `gdb toto`
  - Puis lancement de l'exécution avec la commande `run`

Exemple d'exécution normale

```
1  coti@abidjan:~$ gdb /bin/true
2  GNU gdb (GDB) 7.0.1-debian
3  Copyright (C) 2009 Free Software Foundation, Inc.
4  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
5  This is free software: you are free to change and redistribute it.
6  There is NO WARRANTY, to the extent permitted by law.
   Type "show copying"
7  and "show warranty" for details.
8  This GDB was configured as "i486-linux-gnu".
9  For bug reporting instructions, please see:
10 <http://www.gnu.org/software/gdb/bugs/>...
11 Reading symbols from /bin/true...(no debugging symbols found)...done.
12 (gdb) run
13 Starting program: /bin/true
14
15 Program exited normally.
16 (gdb)
```

Exemple de programme qui plante

Voici un programme incorrect qui va planter sur une erreur de segmentation :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fonction(){
5          int* tab = NULL;
6          int i;
7          for( i = 0 ; i < 40 ; i++ ) {
8                  tab[i] = i;
9          }
10 }
11
12 int main(){
13         fonction();
14         return EXIT_SUCCESS;
15 }
```

Exécution du programme qui plante

```
1  coti@abidjan:~$ gdb plante
2  GNU gdb (GDB) 7.0.1-debian
3  Copyright (C) 2009 Free Software Foundation, Inc.
4  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
5  This is free software: you are free to change and redistribute it.
6  There is NO WARRANTY, to the extent permitted by law.
   Type "show copying"
7  and "show warranty" for details.
8  This GDB was configured as "i486-linux-gnu".
9  For bug reporting instructions, please see:
10 <http://www.gnu.org/software/gdb/bugs/>...
11 Reading symbols from /users/coti/plante...done.
12 (gdb) run
13 Starting program: /users/coti/plante
14
15 Program received signal SIGSEGV, Segmentation fault.
16 0x080483af in fonction () at plante.c:8
17 8                 tab[i] = i;
18 (gdb)
```

Afficher la pile d'appels de fonctions

La **pile d'appels des fonctions** est la suite de fonctions et procédures qui se sont appelées pour arriver à un point donné de l'exécution d'un programme. Il peut être intéressant de l'examiner pour trouver **où la fonction dans laquelle on se trouve a été appelée**.

On peut l'afficher dans gdb grâce à la commande **backtrace** (`bt`)

```
1 (gdb) bt
2 #0  0x080483af in fonction () at plante.c:8
3 #1  0x080483c5 in main () at plante.c:13
4 (gdb)
```

Ici

- On se trouve dans la fonction `fonction()`, ligne 8 du fichier `plante.c`
- Cette fonction a été appelée par la fonction `main()` à la ligne 13 du fichier `plante.c`

Affichage de la valeur d'une variable

On peut afficher la valeur d'une variable au moment où on l'examine en utilisant la commande **print**

```
1 (gdb) print i
2 $1 = 0
3 (gdb) print tab
4 $2 = (int *) 0x0
5 (gdb) print tab[i]
6 Cannot access memory at address 0x0
```

Ici on voit que :

- on a essayé d'accéder à `tab[i]` pour `i` valant 0
- le pointeur `tab` a la valeur `NULL`
- on ne peut pas accéder à la mémoire correspondant à `tab[0]`

Afficher les instructions suivantes et précédentes

On peut examiner quelques lignes du code source situées autour de l'instruction sur laquelle le programme a planté grâce à la commande **list**

```
(gdb) list
3
4 void fonction(){
5           int* tab = NULL;
6           int i;
7           for( i = 0 ; i < 40 ; i++ ) {
8                   tab[i] = i;
9           }
10 }
11
12 int main(){
```

Déplacements dans la pile d'appels des fonctions

On peut remonter dans la pile d'appels de fonctions vers la fonction appelante grâce à la commande **up**

```
1  (gdb) up
2  #1  0x080483c5 in main () at plante.c:13
3  13 fonction();
4  (gdb) list
5  8                   tab[i] = i;
6  9           }
7  10 }
8  11
9  12 int main(){
10 13 fonction();
11 14 return EXIT_SUCCESS;
12 15 }
```

On peut retourner dans la fonction en descendant d'un niveau dans la pile en utilisant la commande **down**:

```
1  (gdb) down
2  #0  0x080483af in fonction () at plante.c:8
3  8                   tab[i] = i;
4  (gdb)
```

Définition d'un point d'arrêt

Un **point d'arrêt** (breakpoint) est un point précis du programme où l'exécution va s'arrêter.

- On l'utilise pour examiner des valeurs lors d'un moment donné de l'exécution (avant que le programme ne plante)

On utilise la commande **break** suivi du numéro de ligne où doit se situer le point d'arrêt.

- Si l'exécutable provient de plusieurs fichiers on précise également le nom du fichier : **fichier.c:ligne**

```
1 (gdb) break 7
2 Breakpoint 1 at 0x80483a1: file plante.c, line 7.
```

Exécution avec des points d'arrêt

On lance l'exécution avec la commande **run** : elle s'arrête sur le point d'arrêt.

```
1 Breakpoint 1, fonction () at plante.c:7
2 7             for( i = 0 ; i < 40 ; i++ ) {
```

On peut regarder la valeur des variables du programme à l'endroit précis du point d'arrêt.

```
1 (gdb) print i
2 $1 = -1208270860
```

On peut continuer l'exécution jusqu'au prochain point d'arrêt avec la commande **continue** :

```
1 (gdb) continue
2 Continuing.
3
4 Program received signal SIGSEGV, Segmentation fault.
5 0x080483b6 in fonction () at plante.c:8
6 8                     tab[i] = i;
```

Affichage et suppression des points d'arrêt

On peut lister les points d'arrêt qui ont été définis sur le programme avec la commande **info breakpoints** :

```
1 (gdb) info breakpoints
2 Num     Type           Disp Enb Address    What
3 1       breakpoint     keep y   0x080483a1 in fonction at plante.c:7
4 breakpoint already hit 1 time
```

On voit que l'on a défini un point d'arrêt et qu'il porte le numéro 1. On peut le supprimer avec la commande **delete numéro** :

```
1 (gdb) delete 1
2 (gdb) info breakpoints
3 No breakpoints or watchpoints.
```