CRO: C langage and programming Roots

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du January 28, 2019

Tanguy Risset

January 28, 2019

Table of Contents

Les pointeurs

- Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale.
- Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée
- C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire.
- Le langage C permet aussi de manipuler directement les cases mémoires à partir de leur adresse.
- Cela permet d'avoir de grandes possibilité à l'exécution (allocation dynamique de mémoire).
- Cela peut aussi résulter en des actions interdites qui plante le programme
- De nombreux langages n'autorisent pas les pointeurs (par exemple Java)

Lvalue

- On appelle Lvalue (modifiable left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une Lvalue est caractérisée par :
  - son adresse, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;
  - sa valeur, c'est-à-dire ce qui est stocké à cette adresse.
- une variable entière est une Lvalue (ex: `i=1`)
- Un tableau ou une chaîne de caractère ne sont pas des Lvalue
- Une Lvalue est rangée à une adresse, on accède à cette adresse par l'opérateur adresse `&`, l'adresse est en général rangée dans un entier (affiché en hexadécimal)
- l'adresse d'une Lvalue n'est **pas** une Lvalue, c'est une constante (on ne peut pas la modifier)

Pointeur

- Un pointeur est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet.
- On déclare un pointeur par l'instruction :
  `type *nom-du-pointeur;`
  où `type` est le type de l'objet pointé.
- Cette déclaration déclare un identificateur, `nom-du-pointeur`, associé à un objet dont la valeur est l'adresse d'un autre objet de type `type`.
- L'identificateur nom-du-pointeur est donc en quelque sorte un *identificateur d'adresse*. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.
- Si on écrit: `int i; i=3; p=&i`
- `p` contient l'adresse de `i` (qui est par exemple la valeur OxBFFFF6C4d).
- Si on regarde a l'adresse OxBFFFF6C4d, on trouve l'entier 3 rangé sur 4 octets
- on y accède par opérateur unaire d'indirection *: `*p` désigne l'objet pointé par `p` c'est à dire la case mémoire à l'adresse OxBFFFF6C4d qui contient la valeur 3.

Valeur et Adresse: exemple

Depuis le passage des adresses sur 64 bits on utilise le format %p pour les addresses

```
int i,j;
i=3;
j=i;
fprintf(stdout,"L'entier i=%d est à l'adresse %p\n",i,&i);
fprintf(stdout,"L'entier j=%d est à l'adresse %p\n",j,&j);
```

affiche

```
L'entier i=3 est à l'adresse 0x7ffd8b130470
L'entier j=3 est à l'adresse 0x7ffd8b130474
```

Valeur et Adresse: exemple

```
#include <stdio.h>

main()
{
  int i = 3;
  int *p;

  p = &i;
  printf("*p = %d \n",*p);
}
```

imprime *p = 3

objet	adresse	valeur
i	OxBFF434000	3
p	OxBFF434004	OxBFF434000
*p	OxBFF434000	3

Valeur et Adresse: exemple

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
}
```

après exécution

objet	adresse	valeur
i	OxBFF434000	6
j	OxBFF434004	6
p1	OxBFF434084	OxBFF434000
p2	OxBFF434092	OxBFF434004

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
}
```

après exécution

objet	adresse	valeur
i	OxBFF434000	3
j	OxBFF434004	6
p1	OxBFF434084	OxBFF434004
p2	OxBFF434092	OxBFF434004

Arithmetique de pointeurs

- La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :
  - l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
  - la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
  - la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.
- Si `i` est un entier et `p` est un pointeur sur un objet de type `type`, l'expression `p + i` désigne un pointeur sur un objet de type `type` dont la valeur est égale à la valeur de `p` incrémentée de `i * sizeof(type)`.

Arithmetique de pointeurs: Exemple

```
main()
{
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = 0x%X \t\n
       p2 = 0x%X\n",p1,p2);
}
```

affiche

p1 = OxBFF434984
p2 = OxBFF434988

```
main()
{
  double i = 3;
  double *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = 0x%X \t\n
         p2 = 0x%X\n",p1,p2);
}
```

affiche

p1 = OxBFF434984
p2 = OxBFF434992

Allocation dynamique

- Avant de manipuler un pointeur il faut l'initialiser. Sa valeur avant initialisation est non determinée.
- Lorsque l'on déclare un pointeur `int *p`, le compilateur reserve un entier pour stocker *l'adresse* vers laquelle pointe `p`, il ne reserve pas de place pour *l'objet* pointé par `p`.
- Soit la place a déjà été réservée lors d'un déclaration précédente: `int i; int *p; p=&i`
- Soit on doit réserver pour `*p` un espace-mémoire de taille adéquate. L'adresse de cet espace-mémoire sera la valeur de `p`.

Attention!!

- Un pointeur pointe:
  - Soit sur une zone mémoire déjà réservée pour une autre variable
  - Soit sur une zone mémoire allouée dynamiquement avec `malloc`

```
main()
{
  int i = 3;
  int *p1;

  p1 = &i;
  [...]
}
```

```
main()
{
  int *p1;

  p1 = (int *)malloc(sizeof(int));

  if (p1==0)
   {
     fprintf(stderr,"Erreur [..]");
     exit(-1);
   }
 [...]
}
```

Segfault..

- Ça, c'est `segmentation fault` direct:

```
main()
{
  int *p1;

  *p1 = 3;
}
```


Allocation dynamique

- Le fait de réserver un espace-mémoire pour stocker l'objet pointé par p s'appelle *allocation dynamique*. Elle se fait en C par la fonction `malloc` de la librairie standard stdlib.h: `malloc(nombre-octets)`
- Cette fonction retourne un pointeur de type `char *` pointant vers un objet de taille `nombre-octets` octets.
- Pour initialiser des pointeurs vers des objets qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un cast. L'argument nombre-octets est souvent donné à l'aide de la fonction sizeof() qui renvoie le nombre d'octets utilisés pour stocker un objet.

```
#include <stdlib.h>
int *p;
p=(int *)malloc(sizeof(int));
*p=1;
```

- Pensez à *tester le résultat* de l'exécution de `malloc` (pas fait ici pour gain de place).

Allocation dynamique: exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int *p;
  int j,i = 3;
  printf("valeur de p avant initialisation = 0x%x\n",p);
  printf("valeur pointée par p avant initialisation = 0x%x\n",*p);
  p = (int*)malloc(sizeof(int));
  printf("valeur de p apres initialisation = 0x%x\n",p);
  printf("valeur pointée par p apres initialisation = 0x%x\n",*p);
  *p = i;
  printf("valeur de p apres affectation = 0x%x\n",p);
  printf("valeur pointée par p apres affectation= %d\n",*p);
}
```

```
valeur de p avant initialisation = 0x80484d0
valeur pointée par p avant initialisation = -2082109099
valeur de p apres initialisation = 0x804a008
valeur pointée par p apres initialisation = 0
valeur de p apres affectation = 0x804a008
valeur pointée par p apres affectation= 3
```


Allocation dynamique: exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int i = 3;
  int j = 6;
  int *p;
  p = (int*)malloc(2 * sizeof(int));
  *p = i;
  *(p + 1) = j;
 printf("p = 0x%X \t *p = %d \t p+1 = 0x%X \t *(p+1) = %d \n",p,*p,p+1,*(p+1));
}
```

```
p = 0xA020008    *p = 3    p+1 = 0xA02000C    *(p+1) = 6
```

Liberation mémoire

- lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement (c'est-à-dire quand on n'utilise plus le pointeur p), il faut libérer cette place en mémoire.
- Ceci se fait à l'aide de l'instruction `free` qui a pour syntaxe `free(nom-du-pointeur)`
- L'instruction `free(p)` libère la mémoire utilisée par l'objet pointé par `p`
- A toute instruction de type `malloc` doit être associée une instruction de type `free`.
- Attention, si la mémoire a été alloué par une déclaration, c'est le compilateur qui se charge de libérer la mémoire lorsque l'on sort de la fonction.
- Il est aussi important d'allouer correctement la mémoire que de libérer correctement la mémoire.

Libération mémoire: Exemple

```
main()
{
  int i = 3;
  int *p;

  p = &i;
  printf("*p = %d \n",*p);
  free(p);
  printf("i = %d \n",i);

}
```

*p = 3
Erreur de segmentation

```
main()
{
  int i = 3;
  int *p;

  p = (int *)malloc(sizeof(int));
  *p=i;
  printf("*p = %d \n",*p);
  free(p);
  printf("i = %d \n",i);

}
```

*p = 3
i = 3

Pointeurs et tableaux

- Tout tableau en C est en fait un pointeur constant. Exemple, `int tab[10]`, `tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau.
- `tab` contient l'adresse de `tab[0]`
- `*tab` contient la valeur de `tab[0]`
- `*(tab+1)` contient la valeur de `tab[1]`
- `p[i]` $\Leftrightarrow$ `*(p + i)`
- On utilise les pointeurs pour manipuler des tableaux dont on ne connait pas la taille à la compilation
- Le véritable sens de la relation entre les tableaux est le suivant: *Une expression de type Tableau-de-type-T devant être évaluée est castée en un pointeur sur son premier élément (à trois exceptions près). Le type résultant est un pointeur sur T*
  exceptions: `sizeof`,`&` et chaines de caractères litéralles.

Pointeurs et tableaux: exemple

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    {
      printf(" %d \n",*p);
      p++;
    }
}
```

```
1
2
6
0
7
```

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    printf(" %d \n", p[i]);
}
```

```
1
2
6
0
7
```

Pointeurs et tableaux: exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int n,i;
  int *tab;

  n=5;
  while (n!=0)
    {
      fprintf(stdout,"Entrer la taille du
                tableau (0 pour finir)\n");
      fscanf(stdin,"%d",&n);

      tab = (int*)malloc(n * sizeof(int));
      for (i=0;i<n;i++)
          tab[i]=i;
      for (i=0;i<n;i++)
          fprintf(stdout,
            "tab[%d]=%d\n",i,*(tab+i));
      free(tab);
    }
```

```
Entrer la taille du tableau
2
tab[0]=0
tab[1]=1
Entrer la taille du tableau
5
tab[0]=0
tab[1]=1
tab[2]=2
tab[3]=3
tab[4]=4
Entrer la taille du tableau
0
```


Pointeur et chaine de caractères

- Une chaîne de caractères était un tableau à une dimension d'objets de type char, se terminant par le caractère nul \0.
- On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type char.
- Cela est souvent utilisé car on connaît rarement à la compilation les tailles de chaines dont on va avoir besoin.

Exemple

```
#include <stdio.h>
main()
{
  int i;
  char *chaine = "chaine de caracteres";

  for (i = 0; *chaine != '\0'; i++)
    chaine++;
  printf("nombre de caracteres = %d\n",i);
}
```


Exemple

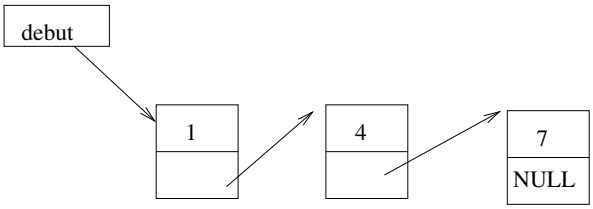
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  int i;
  char *chaine1, *chaine2, *res, *p;

  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((1+strlen(chaine1) + strlen(chaine2)) * sizeof(char)
  p = res;
  for (i = 0; i < strlen(chaine1); i++)
    *p++ = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++)
    *p++ = chaine2[i];
  *p='\0';
  printf("%s\n",res);
}
```

Pointeur et structure

- Contrairement aux tableaux, les objets de type `struct` en C sont des Lvalues.
- Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure.
- On utilise énormément l'association des structures et des pointeurs pour définir des structures de données complexes dynamiquement
- On utilise en particulier beaucoup de *pointeurs sur des structures*

Structures auto-référencées

- On a souvent besoin en C de modèles de structure dont un des membres est un pointeur vers une structure de même modèle.
- Cette représentation permet en particulier de construire des listes chaînées.
- Un élément de la liste chaînée est une structure appelée qui contient la valeur de l'élément et un pointeur sur l'élément suivant.
- Le dernier élément pointe sur la liste vide `NULL`.



Liste d'entier

- On définit le modèle des éléments et le type des éléments :

```
struct model_elem
{
  int val;
  struct model_elem *suivant;
};
typedef struct model_elem ELEMLIST;
```

- on définit un type pour la "tête de liste" contenant certaines informations et permettant de pointer sur le premier élément

```
struct list_mod
{
  int nbElem;
  ELEMLIST *premier ;
} ;
typedef struct list_mod LIST;
```

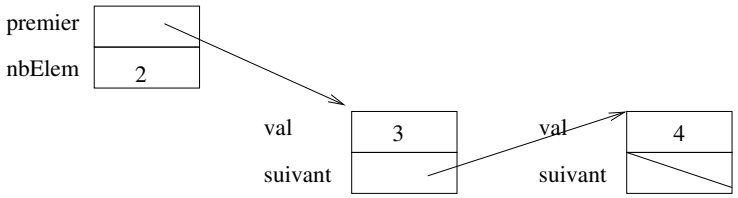

Exemple de manipulation de liste

- Création de liste :

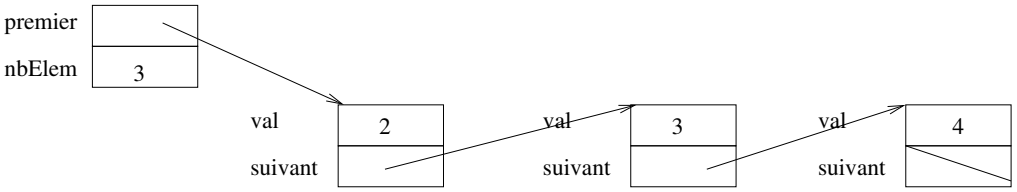
```
struct model_elem
{
  int val;
  struct model_elem *suivant;
};
typedef struct model_elem ELEMLIST;
```

```
 LIST* InitList() /*initialise une liste*/
{
LIST *liste1;

 liste1=(LIST *)malloc(sizeof(LIST));
 liste1->nbElem=0;
 liste1->premier=(ELEMLIST *)0;
 return(liste1);
}
```

Exemple de manipulation de liste



- Ajout d'un élément : avant
- Ajout d'un élément (valeur 2) : après



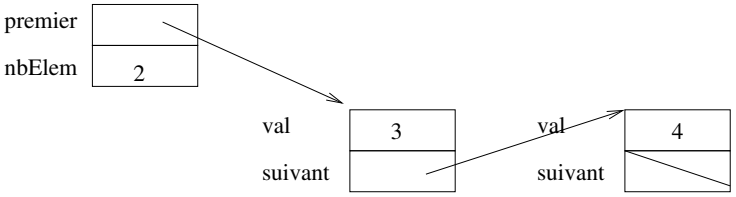
Exemple de manipulation de liste

- Ajout d'un élément en début de liste:

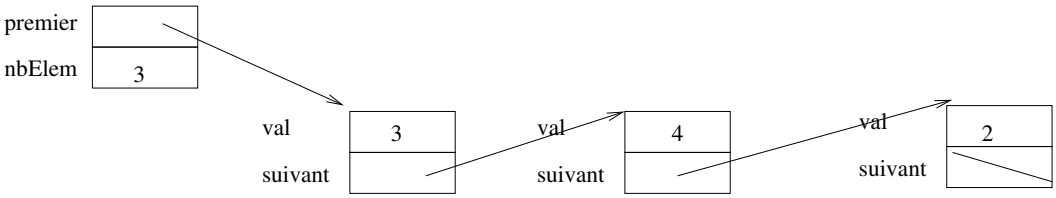
```
/*ajouter un element en premier sommet */
int AjouterElem (LIST *liste, int valeur)
{ ELEMLIST *newelem;
 void *newval;

 newelem=(ELEMLIST *)malloc(sizeof(ELEMLIST));
 if (newelem==0)
   {
     fprintf(stderr,"AjouterElem: plus de place mémoire");
     exit(-1);
   }
 newelem->val=valeur;
 newelem->suivant=liste->premier;
 liste->premier=newelem;
 liste->nbElem=liste->nbElem+1;
 return(0);
}
```

Exemple de manipulation de liste

- Ajout d'un élément en fin de liste: avant



- Ajout d'un élément en fin de liste (valeur 2) : après



Exemple de manipulation de liste

- Ajout d'un élément en fin de liste:

```
/*ajouter un element en fin de list */
int AjouterElemEnFin (LIST *liste, int valeur)
{ ELEMLIST *newelem, *visitor;
 void *newval;

 newelem=(ELEMLIST *)malloc(sizeof(ELEMLIST));
 if (newelem==0)
   {fprintf(stderr,"AjouterElem: plus de place mémoire");
     exit(-1);}
 newelem->val=valeur;
 newelem->suivant=NULL;
 if (liste->premier==NULL)
     liste->premier=newelem;
 else
   {
     visitor=liste->premier;
     while (visitor->suivant!=NULL) visitor=visitor->suivant;
     visitor->suivant=newelem;
   }
 liste->nbElem=liste->nbElem+1;
 return(0);
}
```

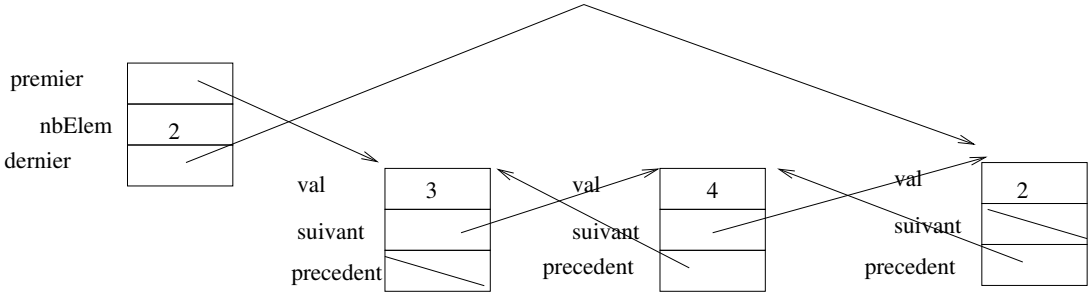
Fonction usuelles sur les liste

- On peut définir les fonctions suivantes:
  - `LIST *initList()`
  - `int ajouterElem(LIST *list,int val)`
  - `int retirerPremierEleme(LIST *list);`
  - `int retirerElemeNumero(LIST *list,int numero);`
  - `int listeVide(LIST *list);`
  - `int nbElements(LIST *list);`
  - `int afficherListe(LIST *list);`

Listes doublement chaînées

- Autre implémentation possible pour les listes:

```
struct model_elem
{
  int val;
  struct model_elem *suivant;
  struct model_elem *precedent;
};
typedef struct model_elem ELEMLIST;
```



Autres structures de données basées sur les listes

- Pile : insertion en début, retrait en début (LIFO)
  - fonction d'accès: empiler(), dépiler(), pilevide(), pilepleine(), etc...
  - implémentée par une liste
- File : insertion en fin, retrait en début (FIFO)
  - fonction d'accès: ajoutelem(), retireelem(), filepleine(), filevide(), fifo()
  - implémentée par une liste doublement chainée.
- Ces structures de données se retrouve dans tous les langages en en algorithmique, on peut évaluer la complexité de chaque opérations et écrire des algorithmes sans se soucier de l'implémentation des opérations de base.

Exemple complet: FIFO d'entier

- Spécification: écrire une bibliothèque permettant de manipuler des FIFO (File) d'entiers de taille `TAILLEMAX` paramétrable statiquement.
- Un dépassement de la taille provoquera un arret du programme.
- Définir un type `FIFO`
- Fournir les fonctions suivantes:
  - `FIFO* InitFifo()`: initialisation de la FIFO
  - `int FifoVide(FIFO*)`: teste si la FIFO est vide
  - `int Ajouter (FIFO*, int)`: ajoute un élément à la FIFO
  - `int Retirer (FIFO*)`: retire un élément de la FIFO et renvoi sa valeur
  - `int HauteurFifo(FIFO*)`: Retourne la taille
  - `void AfficherFifo (FIFO*)`: Affiche le contenu

FIFO: structuration en module

- Proposition d'architecture:
  - Un fichier `fifo_type.h` qui contient la définition du type `FIFO`
  - On peut faire un module `fifo` (fichier `fifo.c` et `fifo.h` contenant les fonctions)
  - Un fichier `test.c` qui teste chacune des fonctions.
  - un fichier `main.c` qui propose d'appeler les test ou de manipuler interactivement une fifo.
- On peut alors créer le makefile.

Makefile (sans le tests)

type FIFO d'entier

- fichier `fifo_type.h`

```
/* Element de fifo */
struct model_elem {
  int elem ;
  struct model_elem* suivant;
};
typedef struct model_elem ELEMFIFO;

/* Type FIFO */
struct model_fifo {
  int nbElem;
  ELEMFIFO* premier;
};
typedef struct model_fifo FIFO;

#define TAILLEMAX 10
```


Gestion d'erreur et InitFifo

```
int error1(char *message) /*affiche l'erreur et quite l'appli*/
{
  fprintf(stderr,"Error: %s\n",message);
  exit(-1);
}

FIFO* InitFifo() /*initialise une fifo*/
{
FIFO *fifo1;

 fifo1=(FIFO *)malloc(sizeof(FIFO));
 if (fifo1==NULL)
     error1("InitFIFO: plus de place mémoire");
 fifo1->nbElem=0;
 fifo1->premier=(ELEMFIFO *)0;

  return(fifo1);
}
```

Ajouter un élément

```
/*ajouter un element au sommet */
int Ajouter (FIFO *fifo, int elem)
{ ELEMFIFO *newelem,*visitor;

 if (fifo->nbElem==TAILLEMAX)
     error1("Ajouter: Taille max de fifo atteinte");

 newelem=(ELEMFIFO *)malloc(sizeof(ELEMFIFO));
 if (newelem==0)
      error1("Ajouter: plus de place mémoire");
 newelem->elem=elem;
 newelem->suivant=fifo->premier;
 //insersion de l'element en tete de FIFO
 fifo->premier=newelem;
 fifo->nbElem=fifo->nbElem+1;

 return(0);
}
```


Afficher FIFO

```
void afficherFifo(FIFO* fifo) /*affiche la fifo d'entier*/
{ELEMFIFO *visitor;

 visitor=fifo->premier;
 while(visitor!=0)
   {
     fprintf(stdout,"|%d|",visitor->elem);
     visitor=visitor->suivant;
   }
 fprintf(stdout,"|--\n");
}
```

Retirer un élément

```
int Retirer(FIFO *fifo) /*retire un element de la fifo*/
{ELEMFIFO *elem1,*visitor,*precedent;
 int val;

 if (fifo->nbElem==0)
   error1("Retirer: fifo vide");
 visitor=fifo->premier;
 if (fifo->nbElem==1) {
     fifo->premier=NULL;
     fifo->nbElem=0;
   }
 else
   {
     precedent=visitor;
     visitor=visitor->suivant;
     while(visitor->suivant!=NULL) {
         visitor=visitor->suivant;
         precedent=precedent->suivant;
       }
     // précédent pointe sur la case suivant de l'avant dernière case
     precedent->suivant=NULL;
     fifo->nbElem=fifo->nbElem-1;
   }
```


Retirer un élément (suite)

```
...
 else
   {
     precedent=visitor;
     visitor=visitor->suivant;
     while(visitor->suivant!=NULL)
       {
         visitor=visitor->suivant;
         precedent=precedent->suivant;
       }
     // précédent pointe sur la case suivant de l'avant dernière case
     precedent->suivant=NULL;
     fifo->nbElem=fifo->nbElem-1;
   }
 //visitor pointer sur le dernier element de la fifo
 val=visitor->elem;
 free(visitor);
 return(val);
}
```

main.c

```
#include <stdio.h>
#include "fifo.h"
int main(int argc,char **argv)
{FIFO *fifo1;
 int elem,elem2,i;

 //test ajouter retirer
 fifo1=InitFifo();
 elem=1;
 Ajouter(fifo1,elem);
 fprintf(stdout,"après ajout de %d\n",elem);
 elem=2;
 Ajouter(fifo1,elem);
 fprintf(stdout,"après ajout de %d\n",elem);
 afficherFifo(fifo1);
 elem2=Retirer(fifo1);
 fprintf(stdout,"après retrait la valeur %d\n",elem2);
 afficherFifo(fifo1);
 //test debordement taille
 for (i=1;i<TAILLEMAX+5;i++) {
      Ajouter(fifo1,elem);
    }
```


Compilation, exécution

```
trisset@hom:~/cours/2005/AGP/cours_tri/fifo$ make
gcc  -c -O3  -g -I.  main.c -o main.o
gcc  main.o fifo.o -o fifo
gcc  main.o -L. -lfifo -o fifoLib
```

```
trisset@hom:~/cours/2005/AGP/cours_tri/fifo$ fifo
/********** test de fifo d'entier ************
après ajout de 1
après ajout de 2
|2||1||--
après retirer la valeur 1
|2||--
Error: Ajouter: Taille max de fifo atteinte
trisset@hom:~/cours/2005/AGP/cours_tri/fifo\$
```

Pointeurs et Tableaux multidimensionnels

- Un tableau à deux dimensions est, par définition, un tableau de tableaux.
- On peut donc le voir comme un pointeur vers un pointeur.
- Considérons le tableau à deux dimensions définit par : `int tab[M][N];`
- Comme dans le cas des tableau 1D, on peut faire la correspondance avec `int **tab`
- `**tab` $\Leftrightarrow$ `tab[0][0]`
- `*(tab[i]+j)` $\Leftrightarrow$ `tab[i][j]`
- `*(*(tab+i)+j)` $\Leftrightarrow$ `tab[i][j]`
- Cependant il existe une différence: lors de la déclaration d'un tableau multi-dimensionnel `int tab[M][N]` le compilateur alloue une zone mémoire contigüe de taille `MxN` (tableau rangé ligne par ligne en C).
- Lors de l'allocation de `int **tab`, l'allocation des colonnes successives n'est pas forcément contigüe

Tableaux multidimensionnels: exemple

```
#define M 2
#define N 3

main()
{
  int **tab;
  int i, j,count=0;
  tab=(int **)malloc(M*sizeof(int *));
  for (i = 0 ; i < M; i++)
    tab[i]=(int *)malloc(N*sizeof(int));
  for (i = 0 ; i < M; i++)
      for (j = 0; j < N; j++)
         tab[i][j]=++count;
  for (i = 0 ; i < M; i++)
      for (j = 0; j < N; j++)
        printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
  for (i = 0 ; i < M; i++)
    free(tab[i]);
  free(tab);
 }
```

exemple

```
Tableau tab déclaré par:
   int tab[2][3] = {{1, 2, 3}, {4, 5, 6}};
tab[0][0]=1
tab[0][1]=2
tab[0][2]=3
tab[1][0]=4
tab[1][1]=5
tab[1][2]=6
*tab[0]=1
*tab[1]=4
*(*(tab+0)+0)=1
*(*(tab+0)+1)=2
*(*(tab+0)+2)=3
*(*(tab+1)+0)=4
*(*(tab+1)+1)=5
*(*(tab+1)+2)=6
Tablean tab2 déclaré par:
 int *tab2; tab2=(int *)tab
tab[0]=134518728
tab2[0]=1
*(tab2+0)=1
*(tab2+1)=2
*(tab2+2)=3
```

```
Tableau tab déclaré par:
int **tab et alloué dynamiquement
tab[0][0]=1
tab[0][1]=2
tab[0][2]=3
tab[1][0]=4
tab[1][1]=5
tab[1][2]=6
*tab[0]=1
*tab[1]=4
*(*(tab+0)+0)=1
*(*(tab+0)+1)=2
*(*(tab+0)+2)=3
*(*(tab+1)+0)=4
*(*(tab+1)+1)=5
*(*(tab+1)+2)=6
Tablean tab2 déclaré par:
int *tab2; tab2=(int *)(*tab)
tab[0]=134520856
tab2[0]=1
*(tab2+0)=1
*(tab2+1)=2
*(tab2+2)=3
```


Tableaux dynamique

- On utilise très fréquement les tableaux dynamiques en C.
- La taille du tableau n'est pas fixée à la compilation.
- Le tableau est déclaré : `int tab[]` pour un tableau d'entier. Cela revient à déclarer un *pointeur* sur un `int`.
- Un tableau de taille `N` est alloué `tab=(int *)malloc(N*sizeof(int))`
- L'utilisation est similaire à celle des tableaux statiques, par exemple:
  - `a=tab[i];`
  - `tab[i]=2;`
- Enfin on libère le tableau lorsqu'on en a plus besoin. `free(tab)`

En résumé

- Equation des pointeurs:
  `&a[i] = a + i`
- Soit un tableau `A` d'un type quelconque, `i` un indice pour les composantes de `A`, alors:
  - `A` désigne l'adresse de `A[0]`
  - `A+i` désigne l'adresse de `A[i]`
  - `*(A+i)` désigne le contenu de A[i]
- `P` un pointeur vers le type des éléments de `A`, et `P = A`, alors:
  - `P` pointe sur l'élément `A[0]`
  - `P+i` pointe sur l'élément `A[i]`
  - `*(P+i)` désigne le contenu de `A[i]`

Exercice

- Que fait la fonction suivante, comment l'utilise-t'on?

```
void fonction(int *xp, int *yp)
{int z;

 z=*xp;
 *xp=*yp;
 *yp=z;
}
```