Programmation 2 :
Consolidation des bases

← ***Suite de : Programmation 1 (bases de la programmation)***

- 7 TD de 2 h
  + 1 TD noté (20pt) de 2h

- 6 TP de 2h avec compte-rendus évalués (total 3 pt)
  + 1 TP individuel noté (17pt) de 2h
  Les TP 1 à 6 doivent être effectués en ***binôme***

Programmation 2 :
PLAN

VI – Les types tableaux (suite)

VII – Types élaborés

- A. Définitions de types (typedef)
- B. Les types symboliques (enum)
- C. Les types structurés (struct)
- D. Les fichiers

VIII – Pointeurs et adresses

- A. Gestion de la mémoire par un exécutable
- B. Allocation dynamique
- C. Retour et paramètres du programme principal (main)

IX – Pour aller plus loin ...

Programmation 2 :
VI – Types tableaux (suite)

- **Tableau à deux dimensions**

  – Grouper **n*p** (constantes) variables de sous un nom unique,
  – Accéder à chaque variable par un **double** indice.
  → *accès au tableau par une (**double**) boucle*
  – Permet de représenter des groupes de valeurs, relations, matrices, etc.

Algorithmique :
Exemple pour n=10 et p=5

```
Pour i de 0 à 9 faire
   Pour j de 0 à 5 faire
      T[i;j]← i+j;
```

Convention : 1er indice (i) → lignes
2e indice (j) → colonnes.

← **colonnes** →

lignes ↑ ↓

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	7	8	9	10	11
3	4	5	6	7	8	9	10	11	12
4	5	6	7	8	9	10	11	12	13
5	6	7	8	9	10	11	12	13	14

Programmation 2 :
VI – Types tableaux (suite)

- **Tableau à deux dimensions : *déclaration***

  type nom[*taille1*][*taille2*]

N.B. En C, tous les éléments d'un tableau sont de même type

Algorithmique :

Exemple pour n=10 et p=5

```
Pour i de 0 à 9 faire
   Pour j de 0 à 5 faire
       T[i;j]← i+j;
```

Convention : 1er indice (i) → lignes
2e indice (j) → colonnes.

Langage C : *exemple*

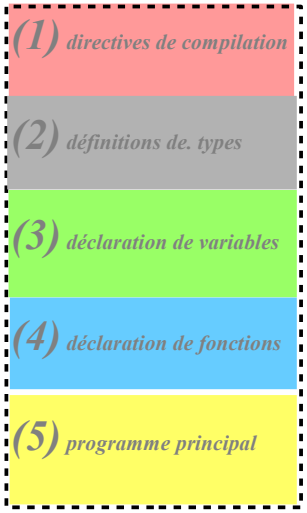
```
#define Nlignes 10
#define Ncolonnes 6

int T[Nlignes][Ncolonnes];
int i,j;

int main() {
for (i=0;i<Nlignes;i++)
  for (j=0;j<Ncolonnes;j++)
     T[i][j]=i+j;
}
```

Programmation 2 :
VI – Types tableaux (suite)

- **Tableau à deux dimensions : *déclaration + initialisation***



Langage C : *exercice*

```
// (zone 3)
int T[2][5]={
      {3,-2,2,-1,-1},
      {5,7,9,11,13}
      };
int U[4][2]={0};
```

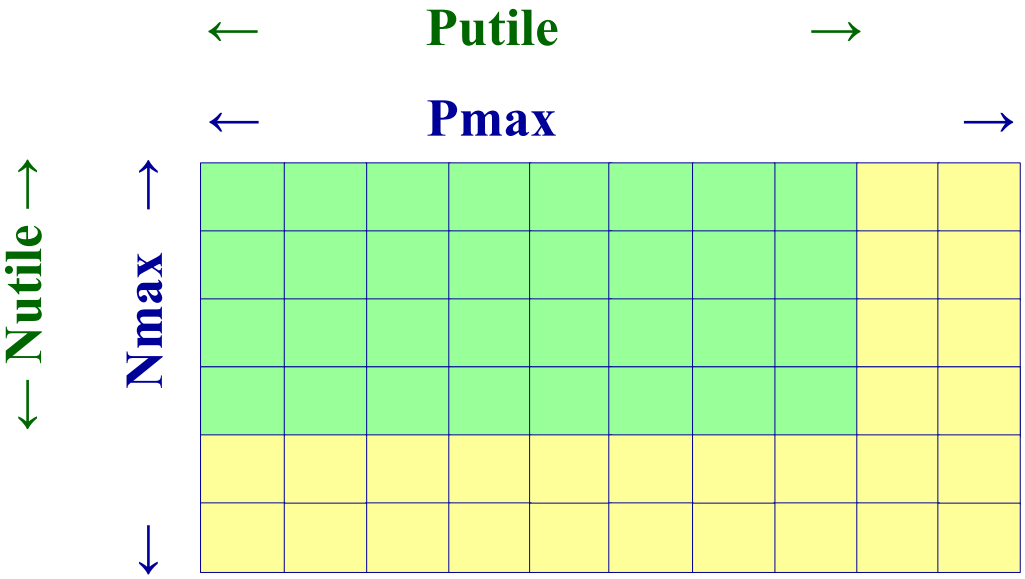
Compléter le programme et tester pour afficher les tableaux T et U (utiliser une double boucle pour chacun)

```
T :
3  -2  2  -1  -1
5   7  9  11  13
U :
0 0 0 0
0 0 0 0
```

Programmation 2 :
VI – Types tableaux (suite)

- **Tableau à deux dimensions : *utilisation partielle***

  – Tailles maximales réservées : Nmax et Pmax,
  – Tailles utiles : Nutile et Putile.



Programmation 2 :
VI – Types tableaux (suite)

- **Tableau de chaines de caractères**

  – Premier indice : nombre de chaines de caractères,
  – Deuxième indice : longueur de la chaine correspondante.

P	r	e	m	i	e	r			
D	e	u	x	i	e	m	e		
T	e	r	c	i	o				
e	t	c	.						
O	K								

N.B. Cela revient à individualiser les tailles des lignes dans un tableau à deux dimensions.

Langage C : *exemple*

```
// (zone 3)
char TS[6][20]={0};
int i;
// (zone 5) :
strcpy(TS[0],"Premier");
strcpy(TS[1],"Deuxieme");
strcpy(TS[2],"Tercio");
strcpy(TS[3],"etc.");
strcpy(TS[4],"OK");
for (i=0;i<6;i++)
   printf("%s\n",TS[i]);
```

Programmation 2 :
VI – Types tableaux (suite)

- ***Pour aller plus loin : tableaux à k>2 dimension***

Algorithmique :	**Langage C** :
k=3 : TroisD[*N1;N2;N3*]	*k=3 :* **`TroisD[`***N1***`][`***N2***`][`***N3***`]`**
k=4 : QuatreD[*N1;N2;N3;N4*]	*k=4 :* **`QuatreD[`***N1***`][`***N2***`][`***N3***`][`***N4***`]`**
k=5 : CinqD[*N1;N2;N3;N4;N5*]	*k=5 :* **`CinqD[`***N1***`][`***N2***`][`***N3***`][`***N4***`][`***N5***`]`**
…	...

Programmation 2 :
VII – Types élaborés

A. Définition de types : `typedef` *definition-du-type* *nom* ;

– Permet de choisir un nom pour un type,
– Très utilisé pour les différents types élaborés.

(1) *directives de compilation*
(2) *définitions de. types*
(3) *déclaration de variables*
(4) *déclaration de fonctions*
(5) *programme principal*

Langage C : *exemples*

```
// (zone 2)
typedef unsigned int natural;
// (zone 3)
natural n;
// (zone 4)
int estpremier(natural n){ … };
```

Quelles différences entre `typedef` et `#define` ?

Programmation 2 :
VII – Types élaborés

B. Les types symboliques : `enum`

`typedef enum {` *Nom1* `, … ,` *Nomk* `}` *NomDuType* `;`

– Permet de nommer une suite finie de constantes symboliques (énumération),
– En C, ces constantes sont représentées par des entiers naturels (0,… , k).

Langage C : *exemples*

```
// (zone 2)
typedef enum {Faux,Vrai} booleen;
typedef enum {Di,Lu,Ma,Me,Je,Ve,Sa} jour;
// (zone 3)
booleen b=Faux ;
booleen Matrice[5][5];
jour j=Lu;
// (zone 4)
booleen estpair(int n) { … }
```

On aura :

Faux=0 et Vrai=1 donc dans la fonction estpair, return Faux <=> return 0.
j=Lu <=> j=1.

Programmation 2 :
VII – Types élaborés – B.

Utiliser un type symbolique pour indicer un tableau

Langage C : *exercice*

```
// (zone 1)
#define Njours 7;
// (zone 2)
typedef enum {Di,Lu,Ma,Me,Je,Ve,Sa} jour;
// (zone 3)
int T[Njours];  // #heures travaillées chaque jour
int i;
// (zone 5)
int main() {
T[Di]=0;
for (i=Lu;i<=Sa;i++) T[i]=8;
// ... puis afficher le total hebdomadaire
}
```

Programmation 2 :
VII – Types élaborés

C. Les types structurés : `struct`

`typedef struct {` *Type1 Var1* **`;`** … **`;`** *typek Vark* **`}`** *NomDuType* **`;`**

– Permet de regrouper des données de types différents,

– Données (champs) encapsulées dans une structure (*cf programmation objet*).

Langage C : ***exercice***

```
// (zone 2)
typedef struct {
   char Nom[20];
   int An;
   int Mois;
   int Jour;
   char Genre;
   } EtatCivil;
// (zone 3)
EtatCivil E;
```

```
// (zone 4)
void AfficherDate (EtatCivil X)
{ printf("%d/%d/%d",X.An,X.Mois,X.Jour); }
// (zone 5)
scanf("%s",E.Nom);
scanf("%d",E.Ann);
scanf("%d",E.Mois);
scanf("%d",E.Jour);
E.Genre='M';
printf("%s né le ",E.Nom);
AfficherDate(E);
```

Compléter et tester

Programmation 2 :
VII – Types élaborés – C.

Pour aller plus loin : affectation groupée des valeurs d'un struct (comme pour un tableau)

Danger : ne pas se tromper dans l'ordre → A EVITER

Langage C : *exemple*

```
// (zone 2)
typedef struct {
   char Nom[20];
   int An;
   int Mois;
   int Jour;
   char Genre;
   } EtatCivil;
```

```
// Déclaration + affectations (zone 3)
EtatCivil E= {
   "Baby", 2019, 12, 11, 'F'
};
```

N.B. *USA : M J A – FR : J M A – JP : A M J*

Programmation 2 :
VII – Types élaborés – C.

Comparaison entre tableaux et types structurés

TABLEAUX	TYPES STRUCTURES
← permettent de regrouper plusieurs variables sous un même nom →	
Variables de **même type**	Variables de **types quelconques**
Accès à chaque **élément** par un **indice** (indiçage du nom du tableau) *Tableau[i]*	Accès à un **champ** par son **nom** (nom de la structure comme préfixe) *Nom_structure.champ*

Programmation 2 :
VII – Types élaborés – C.

Composer des types élaborés : structures incluantes

– Pour une programmation de plus haut niveau,
– Pour se rapprocher de l'algorithmique (*vision plus sémantique*).

Langage C : *exercice*

```
// (zone 2)
typedef enum {
   Inconnu, JAN, …, DEC } Mois;
typedef struct {
   int A; Mois M; int J; } Date;
typedef struct {
   char Nom[20];
   Date Naissance;
   char Genre;
   } EtatCivil;
```

```
// (zone 3)
EtatCivil E;
Date D;
Mois M;
// (zone 4)
..
// (zone 5)
// Saisir votre état-civil
// ...puis l'afficher
// (quel intérêt d'avoir JAN=1?)
```

Compléter et tester

Programmation 2 :
VII – Types élaborés – C.

Composer des types élaborés : tableaux de structure

– Tableau donc chaque élément a le même type structuré
(et comporte donc plusieurs champs)

Langage C : *exemple*

```
// (zone 2)
typedef struct {
   char Nom[20];
   char Prenom[30];
   int NoteTD;
   int NoteTP;
   } UE;
// (zone 3)
UE Prog[64];
int i;
```

```
// (zone 5)
for (i=0;i<64;i++) {
  printf("Nom ? Prénom ? ") ;
  scanf("%s",Prog[i].Nom);
  scanf("%s",Prog[i].Prenom);
  printf("Note TD ? ");
  scanf("%d",Prog[i].NoteTD);
  printf("Note TP ? ");
  scanf("%d",Prog[i].NoteTP);
 }
```

Programmation 2 :
VII – Types élaborés

D. Les fichiers

– Permet l'accès **séquentiels** aux fichiers,
– Utilise les supports permanents de l'ordinateur.

Algorithmique :

Principales opérations de base :

– **fichier F** : F est défini par une chaine de caractères précisant où est le fichier (*exp. F←"~/fichier"*) et d'une position dans le fichier,
– **ouvrirNouveau(F)** : crée un fichier F (ou efface son contenu si F existe déjà),
– **ouvrirLecture(F)** : ouvre le fichier F en lecture et se place au début du fichier F,
– **ouvrirEcriture(F)** : ouvre le fichier F en écriture et se place à la fin du fichier F,
– **estouvert(F)** : vérifie que le fichier F est bien ouvert,
– **lire(F,x)** : place dans x la valeur à la position courante dans le fichier F (et passe à la position suivante dans F),
– **ecrire(F,x)** : place la valeur de x à la fin du fichier F,
– **estfini(F)** : indique si on pointe sur la fin du fichier F,
– **fermer(F)** : ferme l'accès au fichier F (tout fichier ouvert doit être finalement fermé).

Programmation 2 :
VII – Types élaborés – D.

Déclarer une variable de type fichier

Algorithmique : *exemple*

fichier F.

Langage C : *exemple*:

// (zone 3)

```
FILE* F;
```

Programmation 2 :
VII – Types élaborés – D.

Créer (ou réinitialiser) un fichier pour écriture

Algorithmique : *exemple*

fichier F : F est défini par :
- le nom du fichier : *F←"~/fichier" ;*
- une position dans le fichier.

ouvrirNouveau(F) : crée un fichier F
- F est vide (***efface contenu si F existait***).
- on se place au début = à la fin de F.

fermer(F) : ferme l'accès au fichier F.

Si **estouvert(F)** alors
…
fermer(F).

Langage C : *exemple*:

```
// (zone 3)
FILE* F=NULL;


// (zone 5)
F=fopen("fichier","w");

if (F!=NULL) {
   …
   fclose(F);
}
```

"w" : write

Tout fichier ouvert doit être finalement refermé

Programmation 2 :
VII – Types élaborés – D.

Ouvrir un fichier existant en écriture

Algorithmique : *exemple*

fichier F : F est défini par :
- le nom du fichier : *F←"~/fichier" ;*
- une position dans le fichier.

ouvrirEcriture(F) :
- ouvre le fichier F en écriture,
- on se place à la fin de F.

fermer(F) : ferme l'accès au fichier F.

Si **estouvert(F)** alors
...
fermer(F).

Langage C : *exemple*:

```
// (zone 3)
FILE* F=NULL;


// (zone 5)
F=fopen("fichier","a");

if (F!=NULL) {
   ...
   fclose(F);
}
```

"a" : append

Tout fichier ouvert doit être finalement refermé

Programmation 2 :
VII – Types élaborés – D.

Ecrire dans un fichier : `fprintf`

même principe avec "a"

Algorithmique : *exemple*

écrire(F,x) :
- place la valeur de x à la fin de F et
- continue de pointer sur la fin de F.

estfini(F) :
- indique si on pointe sur la fin de F.

```
F← "fichier";
ouvrirEcriture(F);
// oubien ouvrirNouveau(F);
Si estouvert(F) alors
   x← "mot1"; écrire(F,x);
   écrire(F,"mot2");
   fermer(F).
```

Langage C : *exercice*

```
// (zone 3)
FILE* F=NULL;
char S[20];

// (zone 5)
F=fopen("fichier","w");
if (F!=NULL) {
   strcpy(S,"mot1");
   fprintf(F,"%s",S);
   fprintf(F,"%s","mot2");
     }
   fclose(F);
}
```

Compléter et tester

Programmation 2 :
VII – Types élaborés – D.

Ouvrir un fichier existant en lecture

Algorithmique : *exemple*

fichier F : F est défini par ::
- le nom du fichier : *F←"~/fichier" ;*
- une position dans le fichier.

ouvrirLecture(F) :
- ouvre le fichier F en lecture (seule)
- on se place au début de F.

fermer(F) : ferme l'accès au fichier F.

Si **estouvert(F)** alors
…
fermer(F).

Langage C : *exemple*:

```
// (zone 3)
FILE* F=NULL;


// (zone 5)
F=fopen("fichier","r");

if (F!=NULL) {
   ...
   fclose(F);
}
```

"r" : read

Tout fichier ouvert doit être finalement refermé

Programmation 2 :
VII – Types élaborés – D.

Lire dans un fichier : `fscanf`

Algorithmique : *exemple*

lire(F,x) :
- place dans x la valeur de la position courante dans le fichier F et
- passe à la position suivante dans F.

estfini(F) :
- indique si on pointe sur la fin de F.

F← "fichier";
ouvrirLecture(F) ;
Si **estouvert**(F) alors
  Tantque non **estfini**(F) faire
    lire(F,x);
  fermer(F).

Langage C : *exercice*

```
// (zone 3)
FILE* F=NULL;
char S[20];

// (zone 5)
F=fopen("fichier","r");
if (F!=NULL) {
  while (!feof(F)) {
    fscanf(F,"%s",S);
    printf("%s\n",S);
    }
  fclose(F);
}
```

"r"

Compléter et tester

Programmation 2 :
VII – Types élaborés – D.

Entrées et sorties dans un fichier

– Le clavier est considéré comme un fichier d'entrée :
`scanf``(`*format*`,`*variables*`)` <=> **`fscanf`**`(`**`stdin`**`,`*format*`,`*variables*`)`
– L'écran est considéré comme un fichier de sortie :
`printf``(`*format*`,`*variables*`)` <=> **`fprintf`**`(`**`stdout`**`,`*format*`,`*variables*`)`

→ ***fscanf et fprintf accepte des entrées ou sorties formatées (typées : %s,%d, …)***
→ ***fscanf et fprintf ont le même comportement que scanf et printf.***

Donc même comportement déroutant possible en C

N.B. *Ni* **`stdin`** *ni* **`stdout`** *n'ont besoin d'être ouvert ou fermé.*

Programmation 2 :
VIII – Pointeurs et adresses

Avant-propos

– Fichier source exemple.c

compiler

gcc exemple.c

– Fichier exécutable a.out

afficher

cat a.out

– …

Interprétation :

– chaine S …

– entier naturel n ...

Langage C : ***exemple.c***

```
#inclure <stdio.h>
Char * S="ceci est une chaine" ;
unsigned int n=('I'*256+'C')*256+'I';
main() { /* ... */ }
```

Affichage : ***a.out***

Programmation 2 :
VIII – Pointeurs et adresses

A. Gestion de la mémoire par un exécutable

– Fichier source édité

⇓ *enregistrer*

– Fichier source sur disque

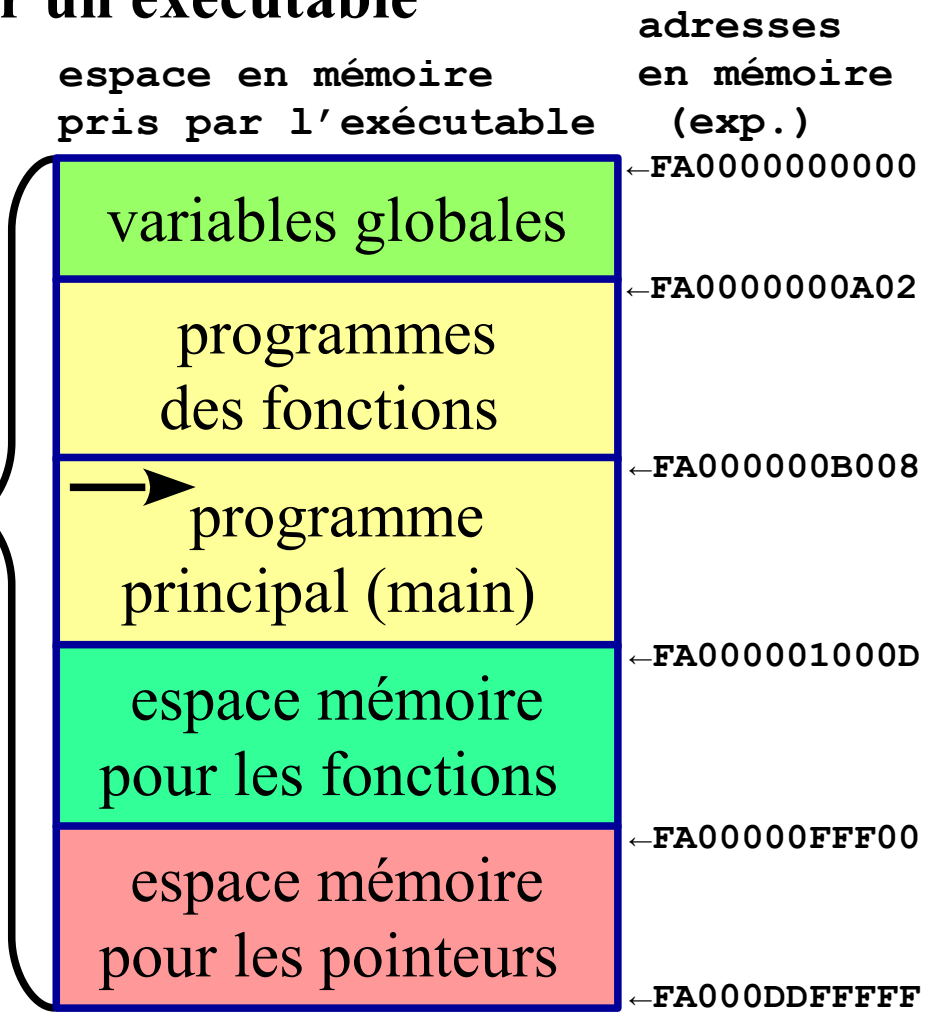
⇓ *compiler*

– Fichier exécutable sur disque

⇓ *exécuter*

– **Fichier exécutable en mémoire**

exp. Ici le programme commence à s'exécuter à l'adresse mémoire FA000000B008 .

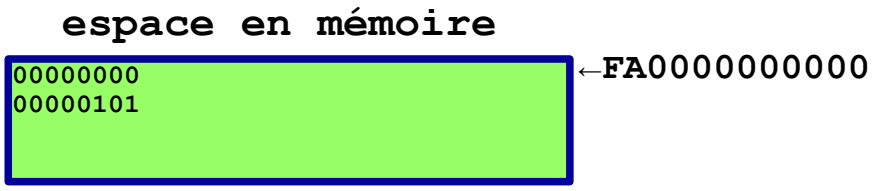


N.B. *Ne pas confondre avec les zones du code source.*

Programmation 2 :
VIII – Pointeurs et adresses – A

A.1. L'exécutable gère les variables

– Variables déclarées :
exp. int n;



– Réaliser une instruction :
exp. n=5;

1. Rechercher l'adresse de n ;
   e*xp. FA0000000000*
2. La taille d'un int est connue ;
   exp. 2 octets (16 bits)
3. Déterminer le codage de 5 sur deux octets ;
   exp. 00000000 00000101
3. Placer cette valeur dans les deux octets de n :
   exp. 00000000 à l'adresse FA0000000000 et
   00000101 à l'adresse FA0000000001.

Programmation 2 :
VIII – Pointeurs et adresses – A.1.

L'exécutable gère les variables : *exercice*

Langage C :

```
// (zone 3)
 int n=5;
 int * p_i=&n;
 char c='a';
 float x=-5.1;
// (zone 5)
 printf("valeur de n : %d\n",n);
 printf("adresse de n : %p\n",&n);
 printf("p est une adresse : %p\n",p_i);
 printf("valeur de *p_i : %d\n",*p_i);
 printf("n est codé sur %d octets\n",sizeof(int));
// ... (faire pareil pour c puis pour x)
```

N.B. scanf utilise aussi l'adresse de la variable

Programmation 2 :
VIII – Pointeurs et adresses – A.1.

L'exécutable gère les variables : vocabulaire

– **Adressage direct** : la valeur de n est atteinte directement

```
int n=5;
```
n → 5.

– **Adressage indirect** : la valeur de n est atteinte avec l'adresse donnée par p.

```
int * p =&n;
```
p → n → 5.

– **Adressage doublement indirect** (…)

```
int * * q=&p;
```
q→ p → n → 5.

– **Référencement** : accès à l'adresse d'une variable

int n=5;

référencement de n : son adresse est donnée par &n, p, ou (*q).

L'accès à la valeur de n est donné par : n, (*p), ou (**q).

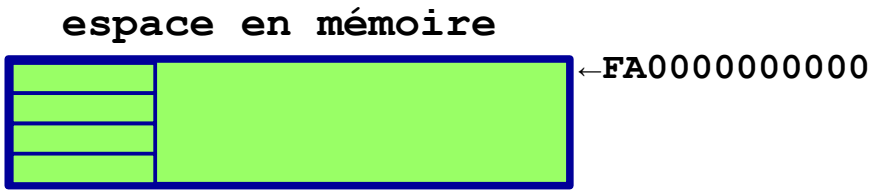
Définir précisément le rôle de l'opérateur * et celui de l'opérateur & (*dans le cadre des pointeurs*)

Ne pas confondre avec * *opérateur de multiplication* ni & avec *opérateur de bit*

Programmation 2 :
VIII – Pointeurs et adresses – A.1.

L'exécutable gère les variables : cas des tableaux

– Variables déclarées :
exp. **`int T[4];`**



1. Réserver quatre espaces mémoire consécutives ;
   e*xp. 2 octets par entier*
   T[0] : FA0000000000 et FA0000000001
   T[1] : FA0000000002 et FA0000000003
   T[2] : FA0000000004 et FA0000000005
   T[3] : FA0000000006 et FA0000000007
2. Gérer chaque T[i] comme une variable de type int.

N.B. Quand on utilise un paramètre int T[] dans une fonction, on lui transmet (recopie) l'***adresse*** de ce tableau T.

Programmation 2 :
VIII – Pointeurs et adresses – A.1.

L'exécutable gère les variables tableau : *exercice*

Langage C :

```
// (zone 3)
 int T[4]={1,2,3,4};
 int i;

// (zone 5)
 for (i=0;i<4;i++)
   printf("T[%d]=%d\n",i,T[i]);
 for (i=0;i<4;i++)
   printf("adresse de T[%d]: %p\n",i,&T[i]);
 for (i=0;i<4;i++)
   printf("T[%d]=%d\n",i,*(T+i));
```

Compléter et tester

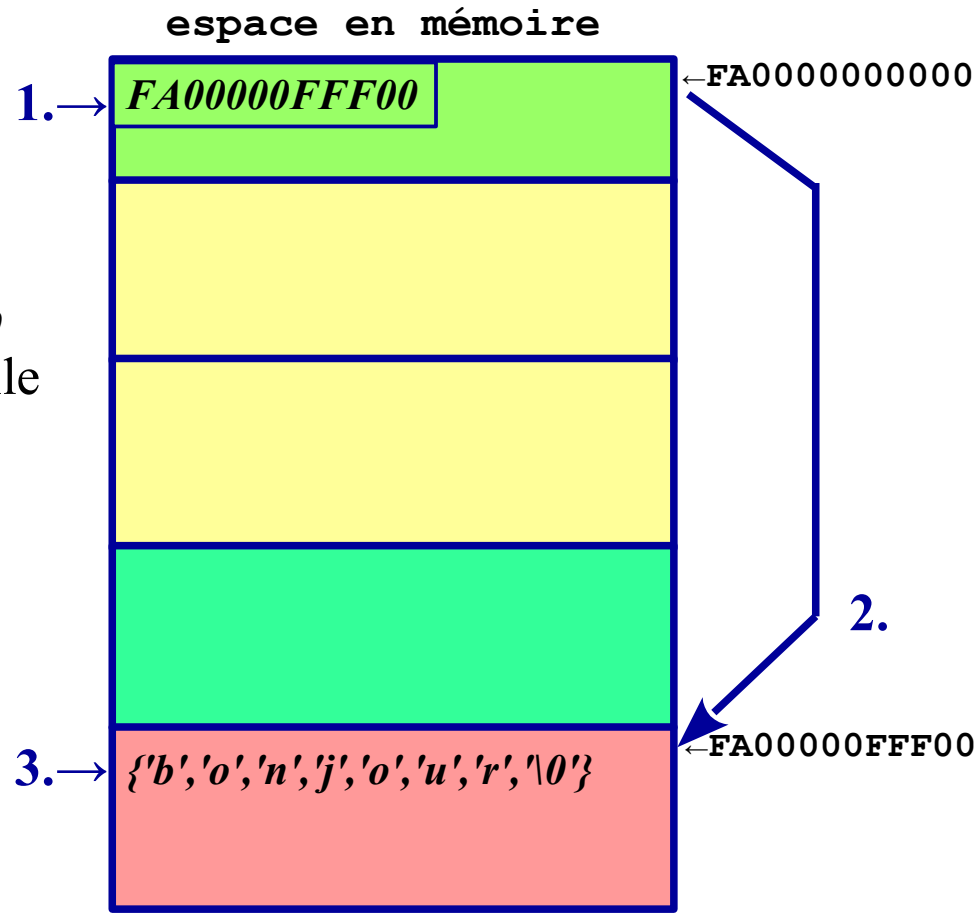
Programmation 2 :
VIII – Pointeurs et adresses – A.1.

L'exécutable gère les variables : cas des chaines de caractères

– Variables déclarées :
exp. char s="bonjour";*

1. Réserver l'espace nécessaire pour un pointeur de type char ;
exp. À l'adresse ***FA0000000000***
placer la valeur ***FA00000FFF00***
2. Réserver un espace libre de taille suffisante où placer la chaine ;
3. A l'adresse pointée, placer la valeur de la chaine ;
exp. Adresse : valeur

FA00000FFF00 : 'b'
FA00000FFF01 : 'o'
FA00000FFF02 : 'n'
FA00000FFF03 : 'j'
FA00000FFF04 : 'o'
FA00000FFF05 : 'u'
FA00000FFF05 : 'r'
FA00000FFF06 : '\0'



Programmation 2 :
VIII – Pointeurs et adresses – A.1.

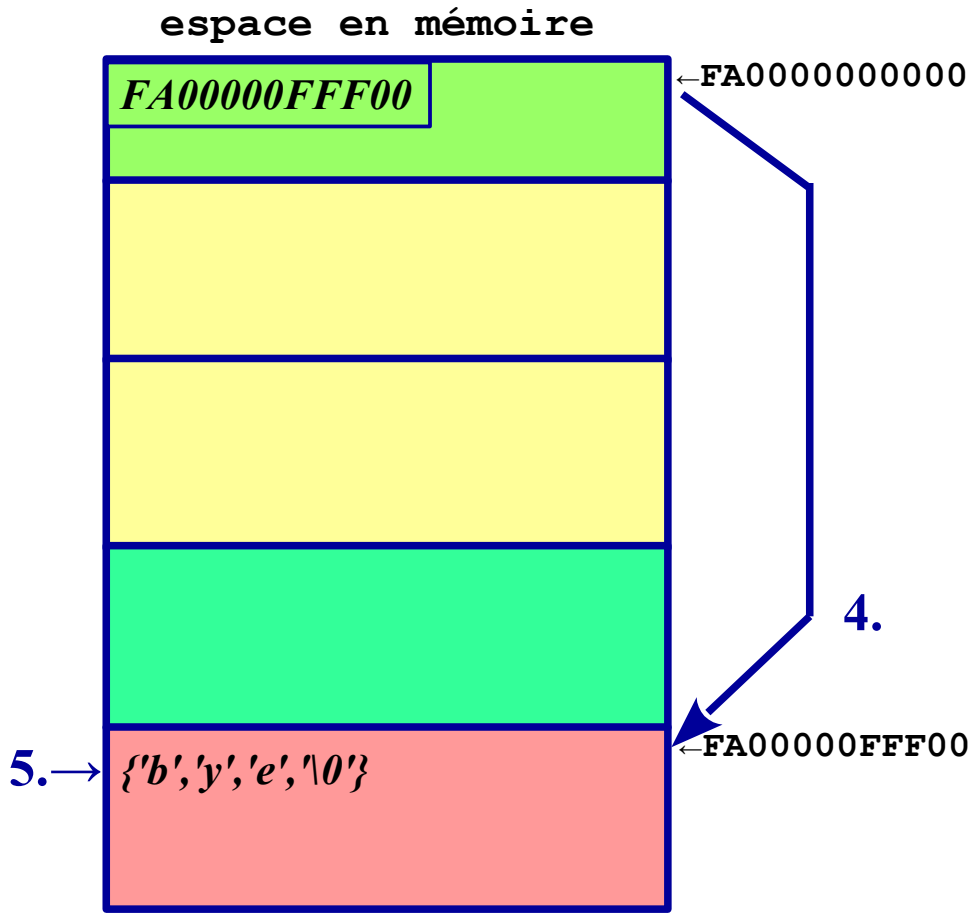
L'exécutable gère les variables : cas des chaines ...

…

– Changement de valeur :
exp. strcpy(s,"bye");

4. Changer si nécessaire l'adresse pointée pour le nouvel espace ;
5. A la (*nouvelle*) adresse pointée, placer la nouvelle valeur de la chaine ;
   exp. Adresse : valeur
   FA00000FFF00 : 'b'
   FA00000FFF01 : 'y'
   FA00000FFF02 : e'
   FA00000FFF03 : '\0'

N.B. *la mémoire libérée peut conserver d'anciennes valeur.*
exp. ***FA00000FFF04 : 'o'***



Programmation 2 :
VIII – Pointeurs et adresses – A.1.

L'exécutable gère les variables chaines : *exercice*

Langage C :

```
// (zone 3)
 char* s="bonjour";
 int i;
// (zone 5)
 i=0;
 while (s[i]!='\0')
   { printf("s[%d]='%c'\n",i,s[i]); i++; }
 i=0;
 while (s[i]!='\0')
   { printf("adresse de s[%d] : %p\n",i,s+i); i++; }
 i=0;
 while (s[i]!='\0')
   { printf("s[%d]='%c'\n",i,*(s+i)); i++; }
```

Compléter et tester

Programmation 2 :
VIII – Pointeurs et adresses – A.

A.3. Passage des paramètres et retour dans une fonction

– Passage d'un paramètre
  – par valeur
  – par adresse

– Retour
  – par valeur
  – par adresse

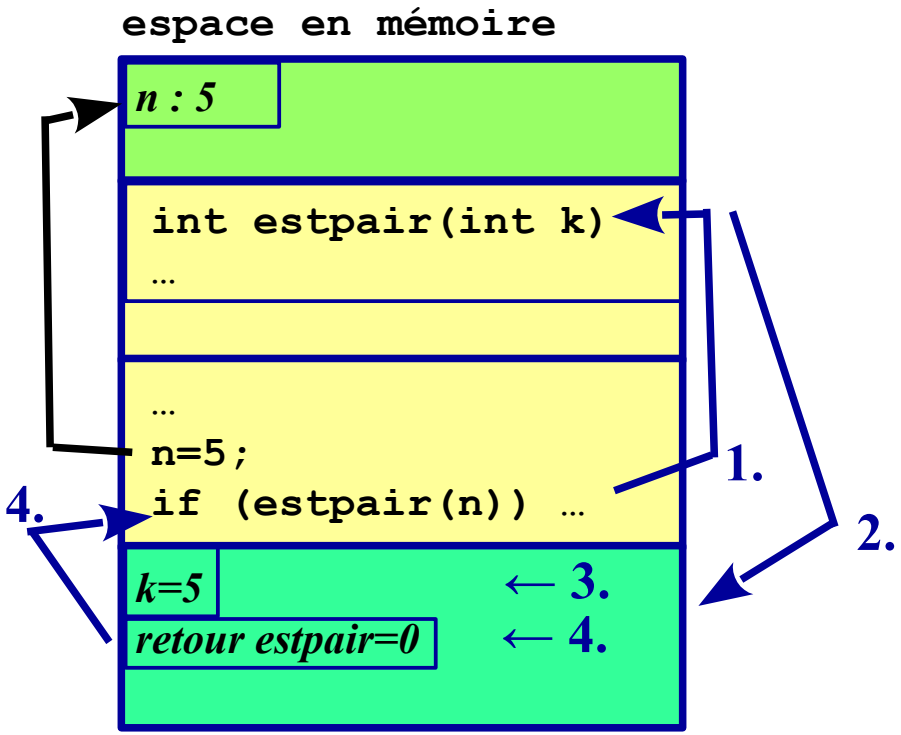
A chaque appel d'une fonction, la valeur des paramètres est recopiée pour la fonction … mis cette valeur peut être un pointeur (copie de l'adresse).

Programmation 2 :
VIII – Pointeurs et adresses – A.

A.2. L'exécutable gère les fonctions :

– Variables déclarées :
exp. int n;
– Fonctions déclarées :
exp. Int estpair(int k);
– Dans le programme :
exp. n=5; if (estpair(n)) ...

1. Trouver l'adresse de estpair ;
2. Réserver l'espace nécessaire
   - pour le paramètre k,
   - pour le retour de la fonction ;
3. Recopier la valeur de n dans k ;
4. Lancer la fonction et récupérer son retour ;
5. Revenir dans le programme et libérer l'espace réservé en 2.

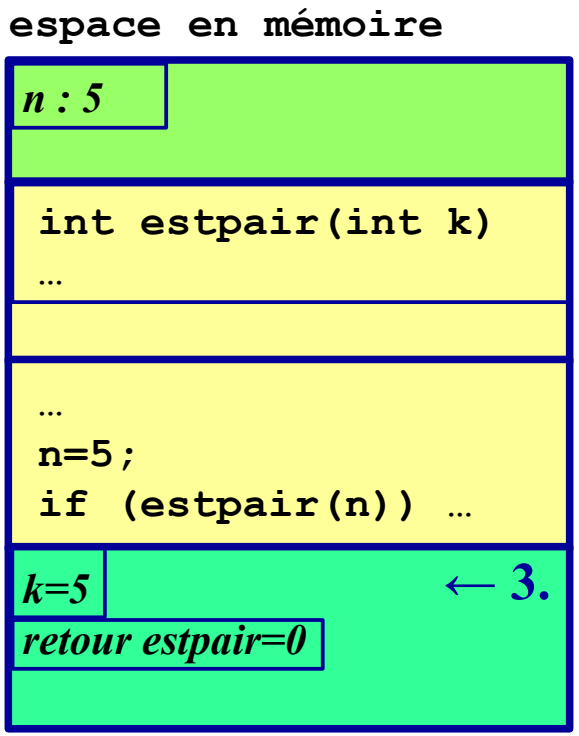


Programmation 2 :
VIII – Pointeurs et adresses – A.3.

Fonction : passage d'un paramètre par valeur

– Variables déclarées :
exp. int n;
– Fonctions déclarées :
exp. int estpair(int k);

3. La valeur de n est recopiée pour l'appel de la fonction ;
→ **la variable n n'est donc pas modifiée par cet appel** (pas d'*effet de bord*).



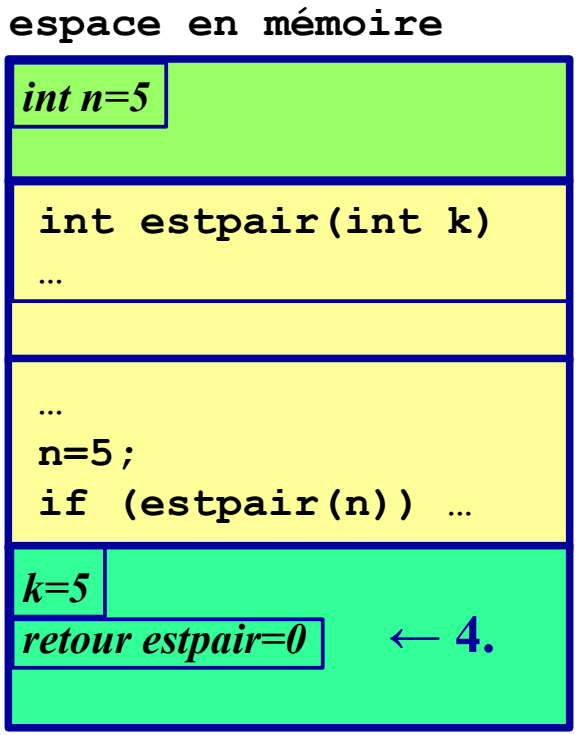
N.B. Ceci est valable pour un paramètre de type structuré

Programmation 2 :
VIII – Pointeurs et adresses – A.3.

Fonction : retour par valeur

– Variables déclarées :
exp. int n;
– Fonctions déclarées :
exp. Int estpair(int k);

4. Le retour d'un appel à la fonction est une valeur;
→ **Cette valeur est disponible pour le programme appelant** (pas d'*effet de bord*).



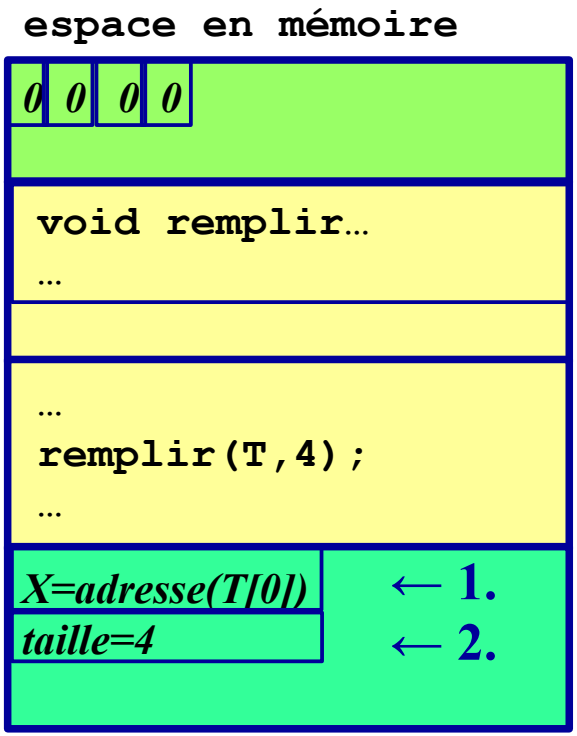
N.B. Ceci est valable pour le retour d'une valeur de type structuré

Programmation 2 :
VIII – Pointeurs et adresses – A.3.

Fonction : passage de paramètre par adresse

– Variables déclarées :
exp. int T[4]={0};
– Fonctions déclarées :
exp. void remplir(int X[],int taille)
{ for (i=0;i<taille;i++) X[i])=2*i+1; }
– Appel
exp. remplir(T,4);

1. La valeur qui est recopiée pour X est la valeur de T[] qui est la valeur d'une adresse.
→ **T sera donc modifié par effet de bord**.
2. La valeur qui est recopiée pour taille est quatre.
→ **pas d'effet de bord pour ce 2e paramètre**.



Programmation 2 :
VIII – Pointeurs et adresses – A.3.

Fonction : passage de paramètre par adresse : *exercice*

Langage C :

```
// (zone 3)
 int T[4]={0};
 int s[3]={0}
 int i;
// (zone 4)
 void remplir(int X[], int taille)
  { for (i=0;i<taille;i++) X[i]=2*i+1; }
 void afficher(int X[], int taille)
  { for (i=0;i<taille;i++) printf("%d ",X[i]); }
 void affecter(char s[]) { strcpy(s,"bonjour"); }
// (zone 5)
 remplir(T,4); afficher(T,4);
 affecter(s); printf("%s\n",s);
```

N.B. Une chaine se comporte comme un tableau de taille variable

Compléter et tester

Programmation 2 :
VIII – Pointeurs et adresses – A.3.

Fonction : retour par adresse

– Le retour d'un appel à la fonction est un pointeur,
– L'objet « adressé » par ce pointeur sera disponible pour le programme,
→ **Fréquemment utilisé avec une allocation dynamique dans la fonction** (*voir plus loin*).

N.B. *Passer un paramètre par adresse permet de le modifier par effet de bord et donc équivaut à un retour de fonction. Utiliser plusieurs paramètres par adresse donne un retour multiple, impossible à réaliser avec return.*

Programmation 2 : VIII – Pointeurs et adresses

B. Allocation dynamique

– Réserver une partie de l'espace mémoire pour les pointeurs, avec la fonction malloc.

type ***** *pointeur* **;**

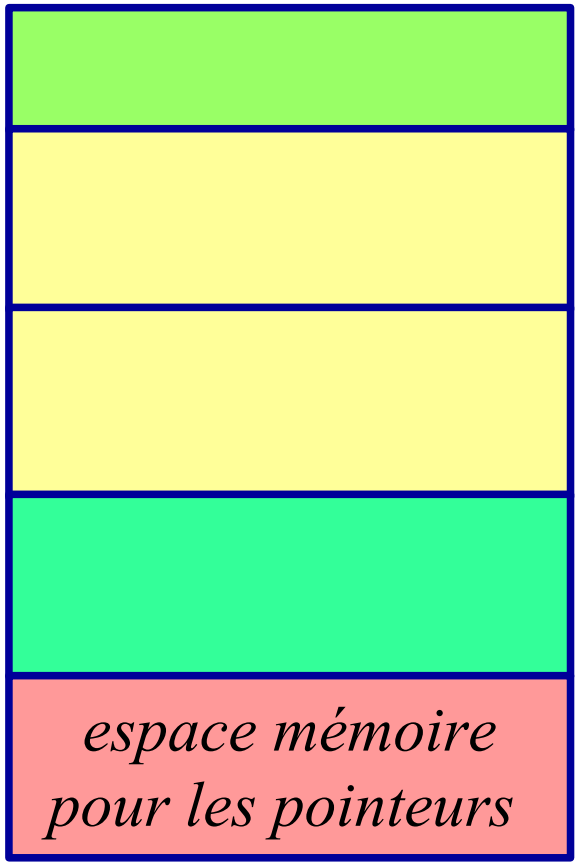
pointeur=**malloc(***nombredoctets***);**

– Utiliser l'adressage indirect, pour avoir accès à cette partie de mémoire réservée.

*****pointeur …

– Libérer la mémoire après usage

free(*pointeur***);**



Rappel :

```
int n=5;      // la variable n a pour adresse &n
int *p=&n;    // p pointe vers la valeur de n
```

Programmation 2 :
VIII – Pointeurs et adresses – B.

Allocation dynamique : *exemple*

Langage C : *allouer un tableau de 10 entiers*

```
#include <stdlib.h>          ← donne accès à malloc et free
#include <stdio.h>
int *T=NULL;                 ← pointeur sur un type entier
int i,n;                        (ne pointant initialement sur rien)
int main() {
 n=10;                       ← on va réserver pour 10 entiers
 T=malloc(n*sizeof(int));    ← demande d'espace pour 10 entiers
 if (T!=NULL) {              ← si l'allocation a réussi ...
   for (i=0;i<n;i++) T[i]=2*i+1; ← (i–1)e valeur du tableau T
   for (i=0;i<n;i++) printf("T[%d]=%d\n",i,T[i]);
   free(T);                  ← libérer la mémoire réellement allouée
 }                              avant la fin du programme
}
```

Programmation 2 :
VIII – Pointeurs et adresses – B.

Allocation dynamique : pointeur et tableau : *exemple*

– **Un tableau est comme un pointeur sur la 1e valeur de ce tableau**

```
int T[10];
```

→ `*T` <=> `T[0]`

→ `T` <=> `&T[0]`

Programmation 2 :
VIII – Pointeurs et adresses – B.

Allocation dynamique : *exercice*

Langage C : *allouer un tableau de 10 entiers*

```
#include <stdlib.h>
#include <stdio.h>
int *T=NULL;
int i,n;
int main() {
 n=10;
 T=malloc(n*sizeof(int));
 if (T!=NULL) {
   for (i=0;i<n;i++) T[i]=2*i+1;
   for (i=0;i<n;i++) printf("T[%d]=%d\n",i,T[i]);
   free(T);
 }
}
```

Tester

Programmation 2 :
VIII – Pointeurs et adresses – B.

Allocation dynamique : cas des chaines de caractères

Langage C :

```
char * s="bonjour";
char t[2]={0};
```

→ *espace alloué automatiquement*

…

```
strcpy(s,"au revoir tout le monde");
strcpy(t,"plus de deux");
```

→ *agrandissement automatique de l'espace alloué*

Une des particularités des chaines de caractères en C...

Programmation 2 :
VIII – Pointeurs et adresses

C. Paramètres et retour de la fonction principale main

– Pour un programme lancé en ligne de commande sous Unix :

– le programme peut retourner au système une valeur entière
Convention : valeur `0` s'il n'y a eu aucun problème.

– le programme peut prendre des paramètres « en ligne de commande »
`./executable param1 … paramk`

Programmation 2 :
VIII – Pointeurs et adresses – C.

C.1. Retour de la fonction principale main

```
int main() {
  …
   return 0;
}
```

← *le retour est toujours de type entier (convention d'Unix et du C)*

← *la valeur 0 indique qu'il n'y a pas eu de problème (convention d'Unix et du C)*

N.B. *L'instruction return du main est facultative dans le code source, on s'en passe si on n'a pas besoin explicitement d'un retour système.*

Programmation 2 :
VIII – Pointeurs et adresses – C.

C.2. Paramètres de la fonction principale main

```
int main(int argCount, char** argVal) {
  …
}
```

– **argCount** donne le nombre d'arguments,
– **argVal** est une liste de argcount chaines de caractères
argVal[0],… argVal[argCount–1].

Exemples :

exec → argCount=1, argVal[0]="exec".
exec prm1 → argCount=2, argVal[0]="exec", argVal[1]="prm1".
exec prm1 prm2 → argCount=4, argVal[0]="exec", argVal[1]="prm1", argVal[2]="prm2".

*N.B. Comme pour tous les paramètre formels, les noms argCount et argVal sont arbitraires. On trouve souvent : int main(int argc, char** argv).*
Le nom de l'exécutable est le premier argument argVal[0]. ArgVal[1] est le 1e paramètre.

Programmation 2 :
VIII – Pointeurs et adresses – C.2.

Exemple d'affichage des paramètres de main

Langage C :

```
#include <stdio.h>

int i;

int main(int argCount, char** argVal) {
 printf("Nom de l'exécutable : %s",argVal[0]);
 printf("Liste des paramètres : \n");
 for (i=0;i<argCount;i++) printf("%s\n",argVal[i]);
}
```

Tester sur des exemples

Programmation 2 :
IX – Pour aller plus loin …

— Fin de l'UE de programmation 2 —

- ***Modules et compilation séparée***
  - *Options de gcc : –E (précompilation), –S (compilation), –c (assemblage), –o (édition de liens)*
  - *Compilation modulaire, fichiers *.h et fichiers *.o*
  - *commande make et fichier makefile*
  - *Classes d'allocation de variables (auto, static, extern)*
  - *Zones de stockage des données (segment de données, pile d'exécution)*

- ***Structuration des données***
  - *Les types union*
  - *Création de types élaborés alloués : liste, pile, file, arbre, etc.*

Programmation 2 : EVALUATION

TD : sur 20 pt

← un TD individuel final noté (sur table) de 2 h

autorisé : une feuille A4 recto-verso écrite de votre main

interdit : tout autre document, tout outil numérique…

N.B. apporter votre carte d'étudiant.

TP : sur 20 pt

← évaluation par binôme des 6 TP de progression : sur 3 pt

← un TP noté final individuel noté de 1h45 : sur 17.

N.B. Programmation 1 et 2 forme un tout indissociable.