

ATELIER DE PROGRAMMATION I



```
return ans;

fcomplex Csqrt(fcomplex z)
{
    fcomplex c;
    float w;
    if ((z.r == 0.0) && (z.i == 0.0))
        c.r=0.0;
        c.i=0.0;
    } else {
        w = sqrt((sqrt(z.r*z.r + z.i*z.i) + z.r)/2);
        c.r = w;
        c.i = (z.i / (2*w));
    }
    return c;
}
```

Plan du cours

-
- Tableaux / Matrices
 - Pointeurs
 - Récursivité



CHAPITRE 1 : TABLEAUX ET MATRICES

Objectifs

- Présenter les structures de données des plus simples aux plus complexes (tableaux, listes, arbres, etc.)
- Donner une maîtrise parfaite du langage de programmation C.

Les tableaux à une dimension

- Déclaration

type nom [dim]

- *type* : définit le type d'élément que contient le tableau. Un tableau en langage C est composé uniquement d'éléments de même type.
- *nom* : est le nom du tableau.
- *dim* : est un nombre entier qui détermine le nombre des éléments du tableau.
- *Exemples*

int compteur[10]; /* le compilateur réserve des places pour 10 entiers, soit 40 octets */

float nombre[20]; /* le compilateur réserve des places pour 20 réels, soit 80 octets */

- *Remarque*

dim est nécessairement une VALEUR NUMERIQUE.

Les tableaux à une dimension

- Accès aux éléments

- Pour accéder à un élément d'un tableau, il suffit de donner le nom du tableau, suivi de l'indice de l'élément entre crochets : `nom[indice]`

- Exemples

- `compteur[2] = 5;`
- `nombre[i] = 6.789;`
- `printf("%d",compteur[i]);`
- `scanf("%f",&nombre[i]);`

- Remarques

- L'indice du premier élément du tableau est 0.
- L'indice est toujours positif.
- L'indice du dernier élément du tableau est égal (`dim- 1`).

Les tableaux à plusieurs dimensions

- Déclaration

type nom [dim1] [dim2]

- Exemples

```
int compteur[4][5];      float nombre[2][10];
```

- Utilisation

Un élément du tableau est repéré par ses indices.

En langage C les tableaux commencent aux indices 0.

Les indices maximum sont donc (dim1-1) et (dim2-1).

Les tableaux à plusieurs dimensions

- Appel

nom[indice1][indice2]

- *Exemples*

int tab[3][4]

tab[0][0] = 5;

Printf ("%d", tab[i][j]);

Scanf ("%f", &tab[i][j]);

<i>Tableau[0][0]</i>	<i>Tableau[0][1]</i>	<i>Tableau[0][2]</i>	<i>Tableau[0][3]</i>
<i>Tableau[1][0]</i>	<i>Tableau[1][1]</i>	<i>Tableau[1][2]</i>	<i>Tableau[1][3]</i>
<i>Tableau[2][0]</i>	<i>Tableau[2][1]</i>	<i>Tableau[2][2]</i>	<i>Tableau[2][3]</i>

Initialisation des tableaux

- Au moment de leur déclaration

Exemples

- *float nombre[4] = {2.67, 5.98, -8, 0.09};*

- *int x[2][3] = { {1, 5, 7} , {8, 4, 3} };*

/ 2 lignes et 3 colonnes */*

Manipulation des tableaux

- Pour la manipulation des tableaux, on utilise les boucles (for ou while ou do ... while).

✓ *Saisie et affichage d'un tableau*

Ecrire un programme permettant de saisir un tableau de réels de taille 10

Manipulation des tableaux

Boucle for	Boucle while
<pre>main() { float nombre[10]; int i; /* saisie du tableau*/ for(i=0; i<10; i++) { printf("nombre[%d] = ", i); scanf("%f", &nombre[i]); } /* affichage du tableau*/ for(i=0; i<10; i++) { printf("nombre[%d] = %f ",i,nombre[i]); }</pre>	<pre>main() { float nombre[10]; int i; /* saisie du tableau*/ i=0; while(i<10) { printf("nombre[%d] = ", i); scanf("%f", &nombre[i]); i++ ; } /* affichage du tableau*/ i=0; while(i<10) { printf("nombre[%d] = %f",i, nombre[i]); i++; } }</pre>

Manipulation des tableaux

- Ecrire un programme permettant de saisir un tableau de 10 entiers, de calculer et d'afficher leur moyenne.

```
int tab[10];
int i,s=0; float moy;
        /* saisie du tableau */
for(i=0;i<10;i++)
{
    printf("tab[%d] = ",i);    scanf("%d", &tab[i]);
}
        /* calcul de la moyenne */
for(i=0; i<10; i++)
{
    s = s+ tab[i];
    moy = s / 10;
}
        /* affichage de la moyenne */
printf("Moyenne du tableau = %f", moy); // ou bien
printf("Moyenne du tableau = %f", float(s/10) );
```

Manipulation des tableaux

- Ecrire un programme permettant de saisir la moyenne de 30 étudiants, de calculer et d'afficher la plus grande moyenne.

```
float moy [30];
int i; float max;
    /* saisie du tableau */
for(i=0; i<30; i++)
{
    printf("moy[%d] = ",i);
    scanf("%f", &moy[i]);
}
max = moy[0] ;
    /* recherche du maximum */
for(i=0; i<30; i++)
{
    if (moy[i] > max)
        max = moy[i];
}

/* affichage du maximum */
printf("Maximum du tableau = %f", max);
```

Saisie et affichage d'un tableau à deux dimensions

- Ecrire un programme permettant de saisir un tableau de réels de taille 5x10

```
float nombre[5][10];
int i,j;
/* saisie du tableau*/
    for( i=0; i<5; i++ )
        for( j=0; j<10; j++ )
        {
            printf("nombre[%d][%d] = ", i,j);
            scanf("%f", &nombre[i][j]);
        }
/* affichage du tableau*/
    for( i=0; i<5; i++ )
        for( j=0; j<10; j++ )
        {
            printf("nombre[%d][%d] = %f ", i, j, nombre[i][j] );
            printf("\n");
        }
```

Addition de deux matrices

- Ecrire un programme qui réalise l'addition de deux matrices A et B de même dimension $N * M$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} a' & b' \\ c' & d' \end{bmatrix} = \begin{bmatrix} a+a' & b+b' \\ c+c' & d+d' \end{bmatrix}$$

```
#include <stdio.h>
main()
{
    /* Déclarations */
    int A[50][50]; /* matrice donnée */
    int B[50][50]; /* matrice donnée */
    int C[50][50]; /* matrice résultat */
    int N, M;      /* dimensions des matrices */
    int I, J;      /* indices courants */
}
```

```
/* Saisie des données */
printf("Nombre de lignes (max.50) : ");
scanf("%d", &N);
```

```
printf("Nombre de colonnes (max.50) : ");
scanf("%d", &M);
printf("*** Matrice A ***\n");
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
    {
        printf("Elément[%d][%d] : ",I,J);
        scanf("%d", &A[I][J]);
    }
printf("*** Matrice B ***\n");
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
    {
        printf("Elément[%d][%d] : ",I,J);
        scanf("%d", &B[I][J]);
    }
```


Addition de deux matrices

```
/* Affichage des matrices */
printf("Matrice donnée A :\n");
for (I=0; I<N; I++)
{
    for (J=0; J<M; J++)
        printf("%7d", A[I][J]);
    printf("\n");
}
printf("Matrice donnée B :\n");
for (I=0; I<N; I++)
{
    for (J=0; J<M; J++)
        printf("%7d", B[I][J]);
    printf("\n");
}
```

```
/* Affectation du résultat de l'addition à C */
for (I=0; I<N; I++)
    for (J=0; J<M; J++)
        C[I][J] = A[I][J]+B[I][J];

/* Edition du résultat */
printf("Matrice résultat C :\n");
for (I=0; I<N; I++)
{
    for (J=0; J<M; J++)
        printf("%7d", C[I][J]);
    printf("\n");
}
return 0;
}
```



CHAPITRE 2 : POINTEURS

Introduction

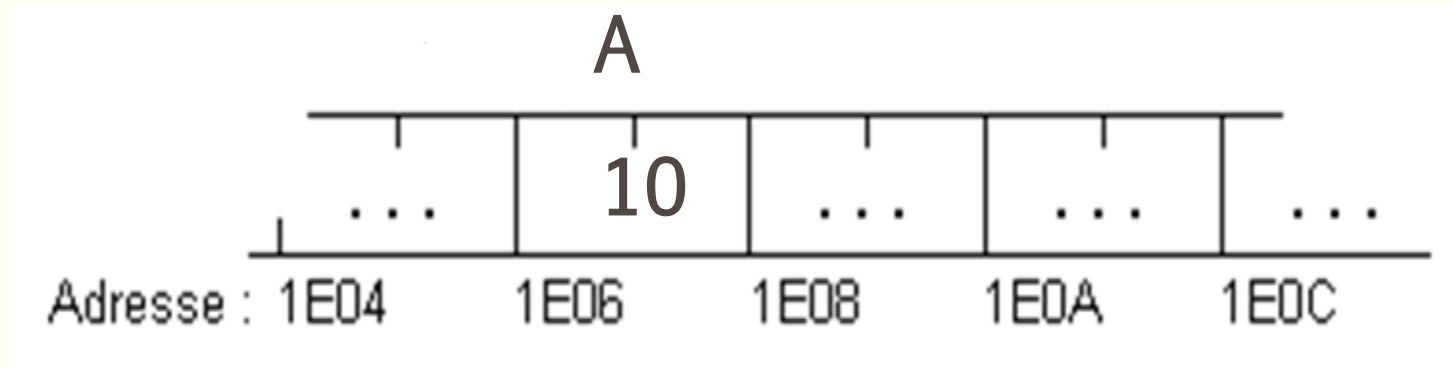
- La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de *pointeurs*, c.-à-d. à l'aide de variables auxquelles on peut attribuer les *adresses d'autres variables*.

I- Adressage de variables

- *Adressage direct* : La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

A: entier

A ← 10



I- Adressage de variables

- *Adressage indirect*

Ne pas utiliser le nom d'une variable A,

→ Copier l'adresse de cette variable dans une variable spéciale P appelée **Pointeur**.

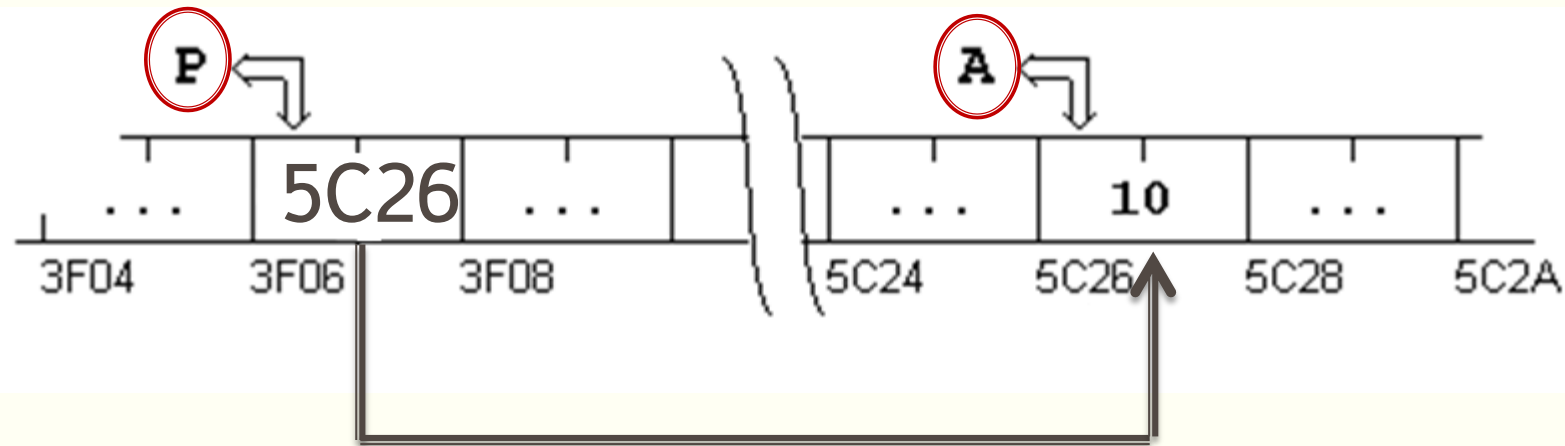
→ Retrouver l'information de la variable A en passant par le pointeur P.

→ Accès au contenu d'une variable en passant par un pointeur qui contient l'adresse de la variable.

I- Adressage de variables

- *Adressage indirect*

int * p : réservation d'un emplacement mémoire pour le pointeur p (case 3F06)
int A =10;



$p = \&A;$

II- Les pointeurs

- *Définition*

Un pointeur est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.

Si un pointeur P contient l'adresse d'une variable A, on dit que '*P pointe sur A*'.

Un pointeur est limité à un type de données. Il peut contenir :

L'adresse d'une variable simple de ce type,

L'adresse d'une composante d'un tableau de ce type.

II- Les pointeurs

- *Les opérateurs de base*

Lors du travail avec des pointeurs, nous avons besoin :

- D'un opérateur '*Adresse de*': & pour obtenir l'adresse d'une variable.
- D'un opérateur '*contenu de*': * pour accéder au contenu d'une adresse.
- D'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

II- Les pointeurs

▪ *Les opérateurs de base*

a) L'opérateur 'adresse de' : &

&< nom variable> : *fournit l'adresse de la variable < nom variable>.*

b) L'opérateur 'contenu de' : *

*< nom pointeur > : *désigne le contenu de l'adresse référencée par le pointeur <nom pointeur> .*

c) Déclaration d'un pointeur :

nom pointeur : Pointeur sur type Déclare un pointeur <nom pointeur> qui peut recevoir des adresses de variables.

Remarque

Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence:

- Un *pointeur* est une variable qui peut 'pointer' sur différentes adresses.
- Le *nom d'une variable* reste toujours lié à la même adresse.

II- Les pointeurs

- *Exemple*

P : pointeur sur entier

A ← 10

B ← 50

- Après les instructions :

P ← &A ➡ P pointe sur A

B ← *P ➡ Le contenu de A est affecté à B

*P ← 20 ➡ Le contenu de A est mis à 20



II- Les pointeurs

- Les opérations élémentaires sur les pointeurs

- ❖ Priorité de * et & : même priorité; ils sont évalués de droite à gauche

- ❖ Exemple : Après l'instruction `P = &X;`

`Y = *P + 1` ↔ `Y = X + 1`

`*P = *P + 10` ↔ `X = X + 10`

`*P += 2` ↔ `X += 2`

`++*P` ↔ `++X`

`(*P)++` ↔ `X++` (les parenthèses sont obligatoires)

- Le pointeur NULL


`int * P / P = 0;` le pointeur P ne pointe nulle part

- Pointeurs sont des variables?

`P1 = P2` / copie le contenu de P2 dans P1 donc P1 et P2 pointent sur le même objet.

III- Pointeurs et tableaux

Adressage des composantes d'un tableau

- le nom d'un tableau représente l'adresse de son premier élément
- `&tableau [0]`  `tableau`
- Le nom d'un tableau représente un pointeur constant sur le premier élément du tableau
- Exemple: soit un tableau A de type `int` et un pointeur p sur `int`

`int A[10] / int *p`

`P = A` équivalente à `P = &A[0]`

Si P pointe sur une composante quelconque d'un tableau, alors `P+1` pointe sur la composante suivante

- `P+i` pointe sur la $i^{\text{ème}}$ composante derrière P et `P-i` pointe sur la $i^{\text{ème}}$ composante devant P.
- `*(P+1)` désigne le contenu de `A[1]`
- `*(P+2)` désigne le contenu de `A[2]`

III- Pointeurs et tableaux

Arithmétique des pointeurs

- *Affectation par un pointeur de même type* : Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction **P1 = P2;** fait pointer P1 sur le même objet que P2
- **Addition et soustraction d'un nombre entier** : Si P pointe sur l'élément A[i] d'un tableau, alors **P+n** pointe sur A[i+n] et **P-n** pointe sur A[i-n]
- **Incrémentation et décrémentation d'un compteur** : Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction
P++; P pointe sur A[i+1]
P+=n; P pointe sur A[i+n]
P--; P pointe sur A[i-1]
P-=n; P pointe sur A[i-n]

III- Pointeurs et tableaux

Arithmétique des pointeurs

Domaine des opérations:

- L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies *à l'intérieur d'un tableau*. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.
- *Seule exception:* Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau)
- Exemple

int A[10];

*int *P;*

P = A+9; / dernier élément → légal */*

P = A+10; / dernier élément + 1 → légal */*

P = A+11; / dernier élément + 2 → illégal */*

P = A-1; / premier élément - 1 → illégal */*

III- Pointeurs et tableaux

Arithmétique des pointeurs

- Soustraction de deux pointeurs

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau*

► **P1-P2** fournit le nombre de composantes comprises entre P1 et P2.

- Le résultat de la soustraction **P1-P2** est **négatif** si P1 précède P2

zéro si $P1 = P2$

positif si P2 précède P1

indéfini, si P1 et P2 ne pointent pas dans le même tableau

- Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

III- Pointeurs et tableaux

Arithmétique des pointeurs

- Comparaison de deux pointeurs
- On peut comparer deux pointeurs par $<$, $>$, $<=$, $>=$, $==$, $!=$.
- La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

1^{ère} Application

Visualiser le contenu de A, B, C, P1 et P2 après chacune des instructions suivantes :

```
main()
{
    int A =1;
    int B = 2;
    int C = 3;
    int *P1, *P2;
    P1 = &A;
    P2 = &C;
    *P1 = (*P2)++;
    P1 = P2;
    P2 = &B;
    *P1--=*P2;
    ++*P2;
    *P1*=*P2;
    A=++*P2**P1;
    P1=&A;
    *P2=*P1/=*P2;
    Return 0;
}
```

III- Paramètres d'une fonction

- 1- Passage des paramètres par valeur

Algorithme PAR_VALEUR

VAR

 X, Y : entier

 Procédure PERMUTER (A, B : entier)

 VAR

 AIDE : entier

 DEBUT

 AIDE ← A

 A ← B

 B ← AIDE

 Écrire ("Dans PERMUTER : A=", A, "B= ",

B)

 FIN

DEBUT (P.P)

 A ← 30

 B ← 40

 Écrire ("Avant appel de PERMUTER : X=", X, "Y = ", Y)

 PERMUTER (X, Y)

 Écrire ("Après appel de PERMUTER : X=", X, "Y = ", Y)

FIN

Exécution :

Avant appel de PERMUTER :

X = 30

Y = 40

Dans PERMUTER :

A = 40

B = 30

Après appel de PERMUTER :

X = 30

Y = 40

➔ X et Y restent échangés.

- Lors de l'appel, les valeurs de X et Y sont copiées dans les paramètres A et B. PERMUTER échange bien le contenu des variables locales A et B, mais les valeurs de X et Y restent les mêmes.
- Pour changer la valeur d'une variable de la fonction appelante, nous allons procéder comme suit :
 - La fonction appelante doit fournir l'adresse de la variable et
 - La fonction appelée doit déclarer le paramètre comme pointeur.

III- Paramètres d'une fonction

- *Passage des paramètres par adresse*

Algorithme PAR_ADRESSE

VAR

X, Y : entier

Procédure PERMUTER (A, B : pointeur sur entier)

VAR

AIDE : entier

DEBUT

AIDE ← *A

*A ← *B

*B ← AIDE

Écrire ("Dans PERMUTER : A=", *A, "B= ",

*B)

FIN

DEBUT (P.P)

X ← 30

Y ← 40

Écrire ("Avant appel de PERMUTER : X=", X,

"Y = ", Y)

PERMUTER (&X, &Y)

Écrire ("Après appel de PERMUTER : X=", X,

"Y = ", Y)

FIN

Exécution :

Avant appel de PERMUTER :

X = 30

Y = 40

Dans PERMUTER :

A = 40

B = 30

Après appel de PERMUTER :

X = 40

Y = 30

Lors de l'appel, les adresse de X et Y sont copiées dans les pointeurs A et B. PERMUTER échange ensuite le contenu des adresses indiquées par les pointeurs A et B.

Remarque

- Par défaut lorsque l'on déclare un pointeur, on ne sait pas sur quoi il pointe. Comme toute variable, il faut l'initialiser.

On peut dire qu'un pointeur ne pointe sur rien en lui affectant la valeur NULL

i : entier

p1, p2 : pointeur sur entier

p1 ← &i

p2 ← NULL

IV. Pointeurs et chaines de caractères

Pointeurs sur char et chaînes de caractères constantes

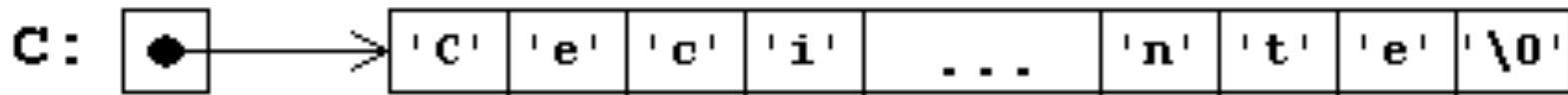
- *Affectation*

On peut attribuer *l'adresse d'une chaîne de caractères constante* à un pointeur sur char:

- Exemple

*char *C;*

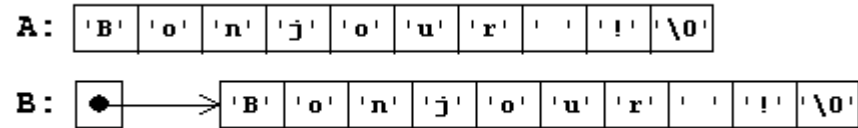
C = "Ceci est une chaîne de caractères constante";



IV. Pointeurs et chaines de caractères

Pointeurs sur char et chaînes de caractères constantes

- Initialisation
- Un pointeur sur `char` peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:
- `char *B = "Bonjour !";`
- Soient les deux déclarations suivantes :
- `char A[] = "Bonjour !";`
- `char *B = "Bonjour !";`
- *A est un tableau* qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison `'\0'`. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.
- *B est un pointeur* qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



V. Tableaux de pointeurs

- Déclaration d'un tableau de pointeurs

*$\langle Type \rangle * \langle NomTableau \rangle [\langle N \rangle]$*

déclare un tableau $\langle NomTableau \rangle$ de $\langle N \rangle$ pointeurs sur des données du type $\langle Type \rangle$.

- ▶ *Exemple* **double** *A[10]; déclare un tableau de 10 pointeurs sur des rationnels du type **double** dont les adresses et les valeurs ne sont pas encore définies.
- ▶ Initialisation : Nous pouvons initialiser les pointeurs d'un tableau sur **char** par les adresses de chaînes de caractères constantes.
- ▶ Exemple: **char** *jour[]={“dimanche”, “lundi”, “mardi”, “mercredi”, “jeudi”, “vendredi”, “samedi”} déclare un tableau **JOUR[]** de 7 pointeurs sur **char**. Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères.

V- Allocation dynamique de mémoire

Pour réserver l'espace mémoire pour un tableau ou une variable quelconque, on utilise souvent une opération d'allocation statique qui est effectué au début d'un programme (section variables), cette manière de réservation présente l'inconvénient que l'espace réservé doit être connu à l'avance (constant)

Exemple

1- A, B : entier

C : réel

2- Type

TAB : Tableau [1..50] : d'entier

V- Allocation dynamique de mémoire

A l'inverse, l'allocation dynamique permet de réserver un espace mémoire de taille variable en milieu d'exécution d'un programme à ce moment, on ne peut pas utiliser les noms pour accéder à cette zone mémoire, on utilisera par suite les pointeurs.

On dispose deux primitives « ALLOUER » et « LIBERER ». La fonction « ALLOUER » permet de réserver un ensemble de cases mémoires et envoyer l'adresse de début de cette zone.

PTR ← ALLOUER (N)

N : nombre d'octet à réserver.

Une fois on n'a plus besoin de cette zone mémoire, on peut libérer avec :

LIBERER (PTR)

V- Allocation dynamique de mémoire

- La fonction `malloc`
- Définie dans la bibliothèque `alloc.h`.
- Exemple :

```
char *pc;
```

```
int *pi,*pj,*pk;
```

```
float *pr;
```

```
pc = (char*)malloc(10);          /* réserve 10 cases mémoire ≡ 10 caractères */
```

```
pi = (int*)malloc(16);          /* réserve 16 cases mémoire ≡ 4 entiers */
```

```
pr = (float*)malloc(24);        /* réserve 24 places en mémoire ≡ 6 réels */
```

```
pj = (int*)malloc(sizeof(int)); /* réserve la taille d'un entier en mémoire */
```

```
pk = (int*)malloc(3*sizeof(int)); /* réserve la place en mémoire pour 3 entiers */
```

V- Allocation dynamique de mémoire

- La fonction **free**
- Définie dans la bibliothèque alloc.h. Elle permet de libérer de la mémoire.

free(pi); /* libère la place précédemment réservée pour pi */

free(pr); /* libère la place précédemment réservée pour pr */

2^{ème} application

- Soit P un pointeur qui 'pointe' sur un tableau A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
```

```
int *P;
```

```
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions:

a) *P+2

b) *(P+2)

d) &A[4]-3

e) A+3

f) &A[7]-P

g) P+(*P-10)

h) *(P+*(P+8)-A[7])

2^{ème} application: correction

$*P+2 \rightarrow 14$

$*(P+2) \rightarrow 34$

$\&A[4]-3 \rightarrow$ l'adresse de la composante $A[1]$

$A+3 \rightarrow$ l'adresse de la composante $A[3]$

$\&A[7]-P \rightarrow$ la valeur (indice) 7

$P+(*P-10) \rightarrow$ l'adresse de la composante $A[2]$

$*(P+*(P+8)-A[7]) \rightarrow$ la valeur 23



CHAPITRE 3 : RÉCURSIVITÉ

Définitions

- Récursivité \approx Récurrence
- Une fonction est définie par récurrence quand, pour définir la valeur de la fonction en un entier donné, on utilise les valeurs de cette même fonction pour des entiers strictement inférieurs.

Définitions

- Une fonction récursive est une fonction qui s'appelle elle-même directement.
- Une fonction réentrante est une fonction qui s'appelle elle-même indirectement via d'autres fonctions.

Définitions

- Une fonction récursive se compose de deux parties:
- La première partie comporte n ($n \geq 1$) appels à elle-même, directs ou non, décomposant le problème avec des données simplifiées.
- La deuxième partie ne comporte pas d'appels récursifs, se chargeant de traiter le problème de base (de données simples) et constituant l'arrêt de la récursivité.

Récurtivité : Appel

- Quant une fonction récursive fait appel à elle-même → une nouvelle zone de données, distincte de celle de la fonction appelante, est créée sur la pile système pour le nouvel appel de la fonction.

Récurtivité : Appel

- Les arguments de l'appel sont recopiés dans cette nouvelle zone de données en plus des variables locales. La zone est libérée au retour de l'appel récursif.

Exemple : la factorielle

/ pré-condition $n \geq 0$ */*

*int **fact** (int n)*

{

if ($n \leq 1$)

return 1;

else

*return $n * \text{fact}(n-1)$;*

}

Analyse d'une fonction récursive

Graphe des appels d'une fonction

C'est un graphe dont les nœuds représentent les appels de la fonction et les arcs représentent les dépendances entre les appels.

Exemple

$$f(n) = n * f(n-1)$$

$f(1) f(2) \dots f(n-1) f(n)$



Analyse récursive d'un problème

- Soit un problème de taille n
- Soit simplifier le problème: transformer la résolution du problème de taille n en fonction de la résolution du même problème de taille $p < n$
- Appréhender le point d'arrêt de la récurrence.

Exemple 1

- Recherche d'un entier E dans un tableau T
- La taille du problème de recherche est liée à la taille du tableau n
- Rechercher dans un tableau de taille n revient à tester $T[\text{begin}]$ avec E puis rechercher sur le reste (tableau de taille $n-1$)
- La recherche s'arrête si $\text{begin} > \text{end}$ ou quand $T[\text{begin}] == E$

Arrêt d'une fonction récursive

- Réduction du problème
- Le problème doit diminuer de taille à chaque appel (les appels récursifs le rendent simple à résoudre)
- Le point d'arrêt: la récurrence doit s'arrêter sur un problème dont la résolution ne nécessite pas d'appel récursif
(**aboutissement**)

Exemple

- Evaluation de la factorielle

```
int fact (int n)
{
  if ((n==0) || (n==1))
    return 1;
  else
    return n* fact (n-1);
}
```

Est elle correcte ?

Evaluation de la factorielle

- **Réduction du problème:** OUI, car à chaque appel l'argument n décroît ($n-1$) donc la taille du problème diminue par conséquence.
- **Point d'arrêt:** NON, la récurrence s'arrête quand n décroît vers 0 ou 1, mais s'il s'agit des éléments négatifs!!! ($n = -5$)
- $f(-5) \rightarrow f(-6) \rightarrow \dots \rightarrow f(-\infty)$ (stack overflow) débordement de la pile système.

Correction d'une fonction récursive

- Invariant d'une fonction récursive

Une condition vérifiée sur l'état des données traitées par la récurrence.

- Le respect de l'invariant : il doit être vérifié pour chaque (appel) de la résolution récursive.
- Exemple: produit des éléments d'un tableau
→ invariant: produit (T, begin, end)

$$\prod_{i=\text{begin}..\text{end}} T[i]$$

Correction d'une fonction récursive

- Respect de l'invariant

1. si $begin = end$

$$\text{Produit}(T, begin, end) = T[end] = \prod_{i=end..end} T[i]$$

2. sinon $T[begin] * \text{Produit}(T, begin+1, end)$

$$= T[begin] * \prod_{i=begin+1..end} T[i]$$

$$= \prod_{i=begin..end} T[i]$$

Exemple

```
float produit (float T[], int begin, int end)
```

```
{
```

```
if (begin == end)
```

```
return T[end]; /*
```

$$\prod_{i=n} T[i] = T[n]$$

```
else
```

```
return T[begin] * produit [T, begin+1, end];
```

```
/*
```

```
}
```

$$\prod_{i=1..n} T[i] = T[1] * \prod_{i=2..n} T[i]$$

Application

- Fibonacci number ?

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$