Cours IUP – PI1A
Algorithmique et Programmation
Année 2004-2005

Programmation avancée en Langage C

Plis fòs ba pengwen là !

Vincent Pagé vpage@univ-ag.fr Université Antilles Guyane

Présentation et évaluation du groupe

Bonjour à tous.

- Vincent Pagé, Mcf en Informatique.
  - vpage@univ-ag.fr
  - Responsable de la première année d'IUP.
- Provenance des étudiants ?
- Qui a un PC (Windows / Linux ?)
- Qui a déjà programmé en C ?
- Qui a déjà programmé sa calculatrice ?
- Qui sait ce qu'est la programmation en assembleur ?

Evaluation du groupe (composants d'un ordinateur)

Qu'est ce qui fait que votre ordinateur est bien ou pas ?

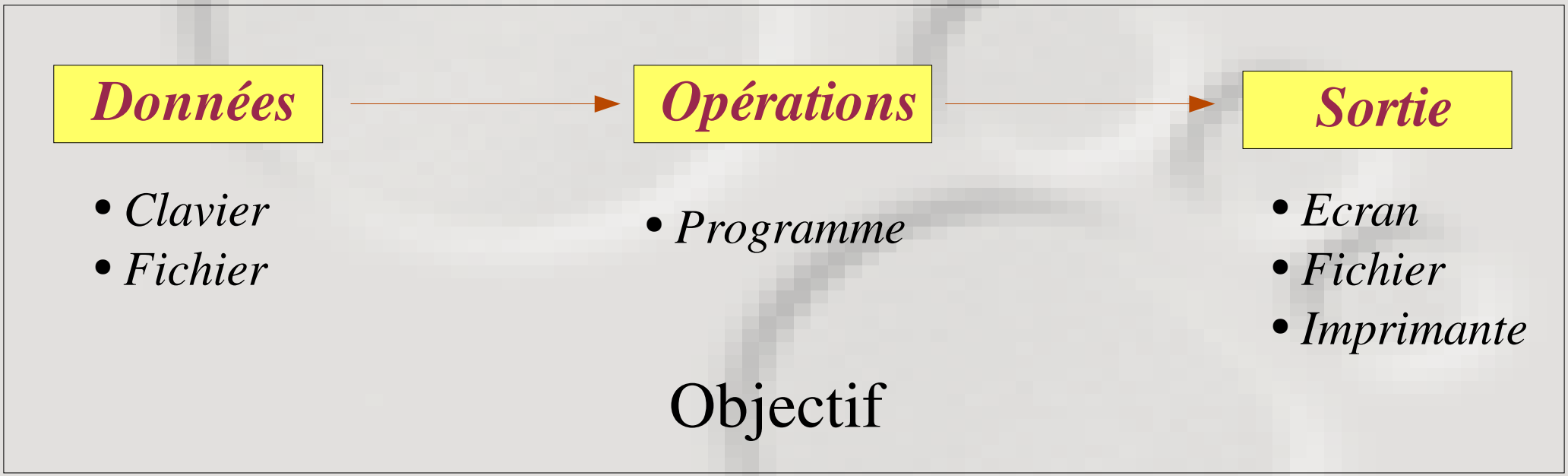
- Vitesse du processeur.
- Mémoire vive.
- Disques durs.

- Annexes : Périphériques, cartes de gestions de périphériques.

Généralités

Que fait un ordinateur ?

DES CALCULS (SIMPLES).



Depuis 1950 (Von Neumann), les opérations sont stockées en mémoire dans l'ordinateur : la machine est programmable.

Généralités (notions de système)

Pour qu'un programme « tourne » (double clic)

- Le programme est chargé « dans » la mémoire de l'ordinateur.
- Le programme s'exécute et travaille sur des données également chargées en mémoire.
- Les sorties sont stockées en mémoire puis redistribuées (écran, fichier, imprimante).

Plus votre ordinateur a de mémoire,
plus il peut faire de grandes choses
(exécuter des gros programmes et travailler sur beaucoup de données)

Notions de compilation

Un programme exécutable est donc un ensemble d'instructions en langage « machine » (proche assembleur)

Pour écrire un programme :

- Décrire les instructions dans un langage (C par exemple).
  C'est le code du programme.
- Transformer ces instructions en langage machine (compilation)
  On obtient un fichier exécutable.

Pour faire tourner le programme :

- Demander au système d'exploitation l'exécution du fichier.

Si vous modifiez votre programme, recompilez avant l'exécution !

Objectifs du cours

Algorithmique et programmation

- Concevoir des algorithmes
  - Combiner des opérations élémentaires pour faire des choses complexes.
- Programmer en Langage C (coder).
- Utiliser du code « tout prêt »
  - Utiliser des fonctions, librairies...
- Organiser les données sous la forme la plus manipulable.
  - Choisir des structures de données (pointeurs, listes...)

Plan du cours

Deux parties

- Jean Luc Henry (algorithmique et structures de données)
- Vincent Pagé (Programmation langage C)
  - Boucles
  - Fonctions
  - Librairies de fonctions
  - Pointeurs et structures de données.
  - Programmation par signaux
  - Projet.

Exercices

Faire un programme qui

- Affiche des X « en triangle »
- Affiche des X « en damier »
- Convertisse des secondes en Heures/Minutes/Secondes.
- Calcule le volume d'une sphère.
- Calcule N tq $\sum_{i=1}^{N} 1/i \leq x$
- Recherche un élément dans un tableau.
- Trouve le plus grand et second plus grand élément d'un tableau.
- Affiche l'intersection de deux tableaux.

Présentation du projet

A partir d'instructions telles que celles que l'on vient de voir, vous allez programmer un jeu vidéo simple.

- Tetris (démo VP)
- Shoot em up (démo VP, Démo Brevignon)
- Casse Brique (Démo VP)

Fonctionnement du cours

- Si vous ne comprenez pas : Arrêtez moi !
- A chaque cours, soyez au point sur ce qui a été dit avant.
- A chaque cours, vous aurez des exercices a faire chez vous.

  (Trouvez un compilateur C.)
- Ne stressez pas sur ces exercices : ce sont des jeux de l'esprit.
- MAIS : faites les exercices (à plusieurs au besoin).
- Notez sur une fiche les erreurs de compilations que vous rencontrez.
- Les choses intéressantes sont complexes, nous allons donc y aller progressivement... Point par point. Passez me voir au besoin.

Exercices du soir

- Faire un programme qui affiche l'intersection de deux tableaux d'entiers positifs.
- Les tableaux auront une taille maximale de 25 entiers.
- L'utilisateur entre autant de nombres qu'il veut dans un tableau lors de la saisie. La saisie est terminée lorsque l'utilisateur entre une valeur négative.
- Les deux tableaux peuvent contenir un nombre différent de valeurs.

Rappels

Structure d'un programme

Commandes pré-processeur
(inclusion de header, constantes)

Programme principal
Début de programme

Déclaration de variables

Instructions

Fin de programme.

Exemple de programme

```
#include <stdio.h>


int main(void)
{

    int a;

    a=5;
    printf("a vaut %d\n",a);

    return 0;
}
```

Les Boucles

Vérifier l'assimilation des boucles :

Correction exercice du soir (saisie d'entiers dans un tableau).

Préférer une solution en

```
while (test==0) {}
```

AVANCEMENT

2 HEURES

Fonctions

Définition : ensemble d'instructions exécutant une même action sur des paramètres qu'on lui passe.

Intérêt : Clarifier la structure d'un programme en regroupant les tâches qui seront répétées au sein du programme.

Exemples :

- Calcul de la factorielle d'un nombre
- Affichage d'un monstre a une position donnée (jeu).

Introduction aux fonctions

Faire un programme qui demande à l'utilisateur un nombre entier et en affiche la factorielle.

Nous allons ensuite en faire une fonction utilisable chaque fois que l'on veut calculer une factorielle.

Fonctions

Exemple de programme avec fonction

```
#include <stdio.h>

int fact (int n)
{
  int i,resu;

  resu=1;
  for (i=2;i<=n;i++)
     resu=resu*i;

  return resu;
}
```

```
int main(void)
{
  int a, calcul;

  a=5;
  calcul = fact (a);
  printf("factorielle de %d vaut %d\n",a,calcul);

  a=6;
  calcul = fact (a);
  printf("factorielle de %d vaut %d\n",a,calcul);

  return 0;
}
```

Fonctions

Paramètres effectif

Dans le programme principal :

On veut utiliser la fonction de calcul de factorielle nommée fact.
Pour calculer cette factorielle, on appelle la fonction « pour » une certaine valeur de a.

La variable a est appelée paramètre effectif de la fonction lors de son appel par le programme principal. Sa valeur doit etre communiquée à la fonction.

Un paramètre effectif a obligatoirement une valeur définie dans le code.

Comment la fonction connait elle la valeur du paramètre effectif ?

Fonctions

Paramètres formels

Dans notre fonction :

Le paramètre formel (ici n) sert à expliquer ce qu'on va faire avec les valeurs que l'on passe à la fonction.
Ici : Servir de limite à la boucle for.

Ce paramètre formel n'a pas de valeur définie dans le code.
C'est une variable locale à la fonction (le programme appelant ne la connaît pas)

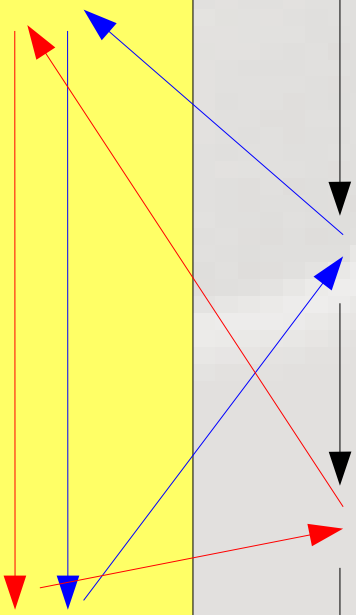
La fonction utilise aussi d'autres variables locales (ici i et resu).
Le programme appelant ne connaît pas ces variables.

Ceci ne nous dit pas comment la fonction utilise la valeur du paramètre effectif qu'on a donné dans le programme principal.

Fonctions

Déroulement des instructions

```
#include <stdio.h>

int fact (int n)
{
  int i,resu;

  resu=1;
  for (i=2;i<=n;i++)
     resu=resu*i;

  return resu;
}
```



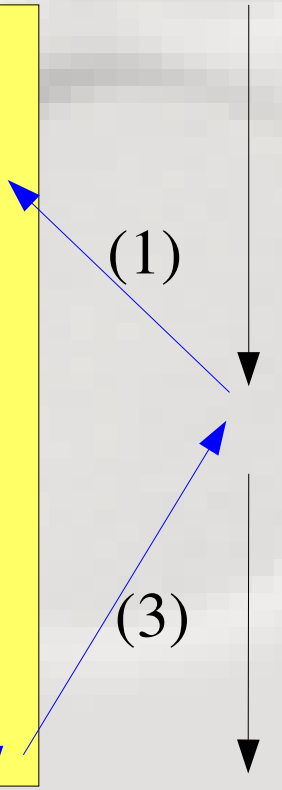
```
int main(void)
{
  int a, calcul;

  a=5;
  calcul = fact (a);
  printf("factorielle de %d vaut %d\n",a,calcul);

  a=6;
  calcul = fact (a);
  printf("factorielle de %d vaut %d\n",a,calcul);

  return 0;
}
```

Fonctions

Déroulement précis des instructions

```
#include <stdio.h>

int fact (int n)
{
  int i,resu;

  resu=1;
  for (i=2;i<=n;i++)
     resu=resu*i;

  return resu;
}
```

```
int main(void)
{
  int a, calcul;

  a=5;
  calcul = fact (a);
  printf("factorielle de %d vaut %d\n",a,calcul);

...
...
```



(1) Chargement du code de la fonction et copie paramètres effectifs -> paramètres formels.

(3) Copie de la valeur de retour dans la variable du sous programme appelant et retour au code du sous-programme appelant

Fonctions

Occupation mémoire

• Les instructions des fonctions et celles du programme principal occupent des espaces mémoires différents.

• Chaque variable nommée occupe un espace mémoire différent, dont la taille dépend du type de variable.

• Lors de l'appel de la fonction, la VALEUR du paramètre effectif est COPIEE à l'emplacement du paramètre formel. La fonction travaille avec cette copie.

Pour le programme principal Pour la fonction fact

Nom formel		A	Calcul			I	N	Resu
Valeur		3	5			3	5	6
Adresse		1112	1137			2178	2212	2240

Fonctions

Passage par valeur

- Lors de l'appel de la fonction, la VALEUR du paramètre effectif est COPIEE à l'emplacement du paramètre formel. La fonction travaille avec cette copie. On parle de PASSAGE PAR VALEUR.

- En langage C, tous les paramètres sont passés par valeur.

Exemple d'application

```
#include <stdio.h>

void bidon (int n)
{
  n=25;
}

int main(void)
{
  int a;
  a=5;
  printf("avant la fonction bidon, a vaut %d\n",a);
  bidon(a);
  printf("après la fonction bidon, a vaut %d\n",a);

  return 0;
}
```

Fonctions

Modification d'une variable du programme principal

Pour qu'une fonction modifie une variable du programme principal, il faut s'arranger pour que la fonction écrive dans l'emplacement mémoire correspondant a cette variable.

Exemple d'application

```
#include <stdio.h>

void modifie (int *n)
{
  *n=25;
}

int main(void)
{
  int a;
  a=5;
  printf("avant la fonction bidon, a vaut %d\n",a);
  modifie(&a);
  printf("après la fonction bidon, a vaut %d\n",a);

  return 0;
}
```

Ici, après la fonction, a vaut bien 25. Ki Bitin ???

Les pointeurs

Manipulation d'adresses mémoire

Rappel : Chaque variable d'un programme qui s'exécute occupe un certain emplacement mémoire.

Exemples :

int : 4 octets
float : 4 octets
char : 1 octet

Ces emplacements mémoires sont repérés par une adresse : le numéro du premier octet ou est stockée la variable. Cette adresse ne varie pas lors de l'exécution du programme.

Type et nom		Int x																															
Valeur en bits				1	0	0	0	1	1	0	1	1	1	0	1	1	0	1	1	1	1	1	0	1	1	1	0	1	1	0	1	1								
Mots en octets		Octet 1								Octet 2								Octet 3								Octet 4									
Adresse		&x																														(&x)+1	

D'habitude, on ne manipule que les variables sans considérations d'adresses. Plus ici !

Adresses et pointeurs

Manipulation d'adresses mémoire

L'adresse d'une variable est accessible en fesant précéder le nom de la variable d'un &.

Si on récapitule sous forme simplifiée :

Type et nom		Int x		Int n
Valeur		5		12
Adresse		&x		&n

Pour manipuler l'adresse d'une variable, il faut un type particulier de données : LES POINTEURS.

Déclaration de pointeurs :

```
int *p_int;          /* Ici p_int contiendra l'adresse d'un entier */
float *p_float;      /* Ici p_float contiendra l'adresse d'un float */
```

Adresses et pointeurs

Manipulation d'adresses mémoire

Exemple :

```
int *p_int;
int a;

a=5;
p_int=&a;       /*p_int contient maintenant l'adresse de a */
```

Nom		A		P_int
Valeur		5		37
Adresse		&a = 37		&p_int

Si x contient une adresse, alors *x est le contenu de cette adresse.

int *p_int;

Cette déclaration peut se lire :

- p_int est un pointeur sur un entier.
- *p_int est un entier.

Adresses et pointeurs

Récapitulatif

Soit x une variable.
  &x est l'adresse de x.

Soit px un pointeur (vaut une adresse)
  *px est la valeur contenue a l'adresse px.

Donc :
Quelle que soit la variable Z, on a :
  *(&Z) = Z

Si P est un pointeur, on a :
  &(*P) = P

Adresses et pointeurs

Exercice

Montrer ce qui se passe en mémoire lors du déroulement du code suivant

```
#include <stdio.h>

void modifie (int *n)
{
  *n=25;
}

int main(void)
```

```
{
  int a;
  a=5;
  printf("avant la fonction bidon, a vaut %d\n",a);
  modifie(&a);
  printf("après la fonction bidon, a vaut %d\n",a);

  return 0;
}
```

On parle alors (abusivement) de PASSAGE PAR ADRESSE

Notez que maintenant, une fonction peut modifier ses paramètres, et vous savez comment/pourquoi.

Exercices du soir

- Expliquez pourquoi on écrit

```
printf("a vaut %d\n",a);

scanf("%d",&a);
```

- Faire une fonction qui échange les valeurs de ses deux paramètres entiers.
- S'en servir pour faire une fonction calculant le complexe conjugué d'un nombre complexe.
- Faire une fonction qui calcule le produit de deux nombres complexes.

AVANCEMENT

4 HEURES

Pointeurs et tableaux

Précisions sur les tableaux

Définition d'un tableau :

- Ensemble de cases mémoires consécutives.
- De taille fixée à la compilation.
- Contenant des données de même type.
- Les valeurs du tableaux sont stockées à une adresse fixe portant le nom du tableau.

int Tab[5];

Nom		Tab[0]	Tab[1]	Tab[2]	Tab[3]	Tab[4]	
Adresse		Tab	Tab+1	Tab+2	Tab+3	Tab+4	

Un TABLEAU est un POINTEUR FIXE
sur la PREMIERE CASE des valeurs qu'il contient

Pointeurs et tableaux

Manipulation de tableaux

Exercice :

Soit la déclaration suivante : `int Tab[5];`

Faites un schéma et montrez moi ce que sont :

- Tab
- Tab + 3
- Tab[2]
- *(Tab +1)
- &(Tab[2])

On a :

- &(Tab[i]) = Tab + i
- *(Tab + i) = Tab[i]

Nom		Tab[0]	*(Tab+1)	Tab[2]	Tab[3]	Tab[4]	
Adresse		Tab	Tab+1	&(Tab[2])	Tab+3	Tab+4	

Tableaux et define

La commande préprocesseur define

D'après vous, quelle est la meilleure façon de coder entre les deux suivantes ?

```
int main (void)
{
  int tab[10],i;

  for (i=0;i<10;i++)
     tab[i]=i;

  for(i=0;i<10;i++)
    printf("tab[%d]=%d\n",i, tab[i]);

  return 0;
}
```

```
#define N 10

int main (void)
{
  int tab[N],i;

  for (i=0;i<N;i++)
     tab[i]=i;

  for(i=0;i<N;i++)
    printf("tab[%d]=%d\n",i, tab[i]);

  return 0;
}
```

Pointeurs et tableaux

Vocabulaire et représentation mnémotechnique

Soit le code suivant :

```
int tab[10];
int a;
int *p;

p=&a;
p=tab+3;
```

- On dit que p pointe sur a ou p pointe sur la troisième case du tableau tab
- On représente ceci par :

*p est la VALEUR au bout de la flèche

Pointeurs et tableaux

Exercices

Exercice : Soit la séquence suivante

```
int tab[10];
int *p, i;

for(i=0;i<10;i++)
    tab[i] = 0;

i=5;
p = tab;
p= tab+i;
```

Décrire le contenu du tableau après ajout des lignes suivantes :

```
(*p)++;
p++;
(*p)++;
(*(p+2))++;
(*(tab+2))++;
p++;
```

Pointeurs

Remarque sur les types de pointeurs

- int *p_int est une variable contenant l'adresse d'un entier.
- float *p_float est une variable contenant l'adresse d'un float.

Ces deux variables contiennent toutes deux l'adresse du premier octet de la valeur qu'elle pointent. Il n'y a pas de différence intrinsèque entre les deux. On les distingue afin que le compilateur puisse s'assurer qu'il n'y a pas de mélange des genres, mais une inversion de type ne génère pas d'erreur à la compilation, seulement un avertissement.

```
int a,*p_int;
float *p_float;

a=5;
p_int=&a;
p_float=p_int; /* ici un avertissement */
printf("a vaut %d\n",*p_float); /* Ici aussi */
```

Doubles Pointeurs

Introduction

- int *p_int est une variable contenant l'adresse d'un entier.

p_int est une variable et possède donc une adresse.

Quel est le type de x=&p_int ???

Il s'agit d'une adresse -> c'est un pointeur.
On a *x=p_int. *x est donc un pointeur sur un entier.
Ou encore *(*x) est un entier.

On a donc le type de x : x est un DOUBLE pointeur sur un entier.
int **x;

Doubles Pointeurs

Exercices

Exercice : Soit la séquence suivante

```
int x=0;y=1;
int *p, **pp;

p=&x;
x+=y;
pp=&p;
```

Décrire le contenu des variables suivantes :
x , *p, **pp

Tableau de pointeurs

Exercices

Exercice : Soit la séquence suivante

```
int *tab[5],a,b,c,i;

a=0;
b=1;
c=2;
tab[0]=&a;
```

```
(*(*(&tab[0])))++;
tab[*(tab[0])]=&c;
*(tab+3)=tab[b];
tab[2]=tab[4]=&c;
(*(tab[2]))++;
for(i=0;i<5;i++
   (*(tab[i]))++;
```

Quelles sont les valeurs des différentes variables ?

AVANCEMENT

6 HEURES

Tableaux bidimensionnels

Introduction

Soit la déclaration suivante : ***int tab[5][3];***

Elle déclare un tableau bidimensionnel.
Ma représentation personnelle en serait :

tab[i][j]

j →

i ↓

Tab[0][0]	Tab[0][1]	Tab[0][2]
Tab[1][0]	Tab[1][1]	Tab[1][2]
Tab[2][0]	Tab[2][1]	Tab[2][2]
Tab[3][0]	Tab[3][1]	Tab[3][2]
Tab[4][0]	Tab[4][1]	Tab[4][2]

Convention dans les indices
lignes d'abord

Tableaux bidimensionnels

Introduction

La représentation en mémoire de ce tableau est un tableau monodimensionnel dans lequel on « colle » les lignes les unes à la suite des autres

Valeur	Tab[0][0]	Tab[0][1]	Tab[0][2]	Tab[1][0]	Tab[1][1]
Adresse	Tab	Tab +1	Tab+2	Tab+3	Tab+4

Pour un tableau défini par tab[N][M],
on a l'egalité suivante tab[i][j] = tab[M*i+j]

Remarque : on peut écrire tab[0][3], ce sera compris comme tab[1][0]

Tableaux

Passage en argument

Si on veut passer un tableau en argument d'une fonction, on passe en fait l'adresse du premier octet (un pointeur).

Problème : Comment la fonction connaît elle la taille du tableau ?

```
#define N 10
#define M 5

void initialise(int tab[N][M])
{
  int i,j;
  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      tab[i][j]=1;
}
```

Solution 1 :
Pas terrible.

Pourquoi ?

```
int main(void)
{
   int montableau[N][M];

  initialise(montableau);
  /* d'autres choses */

  return 0;
}
```

Tableaux

Passage en argument

Sinon, on fait passer un pointeur simple
(un tableau multi-dimensionnel est un pointeur simple)
et les tailles du tableau.
Le compilateur va vous avertir mais c'est correct.
La traduction de tab[i][j] est alors a votre charge.

```
#define N 10
#define M 5

void initialise(int *tab, int n, int m)
{
  int i,j;
  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      tab[i*m+j]=1;
}
```

Solution 2 :

Mieux
Mais tab[i][j]
était pratique...

```
int main(void)
{
   int montableau[N][M];

  int dimi=3,dimj=4;
  initialise(montableau, dimi,dimj);
  /* d'autres choses */

  return 0;
}
```

Tableaux

Passage en argument

Sinon, on fait passer un tableau 2D trop grand et on n'utilise que le nécessaire.

```
#define N 10
#define M 5

void initialise(int tab[N][M], int n, int m)
{
  int i,j;
  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      tab[i][j]=1;
}
```

Solution 3 :

Presque bon
Mais espace mémoire grand

...A suivre...

```
int main(void)
{
   int montableau[N][M];

  int dimi=3,dimj=4;
  initialise(montableau, dimi,dimj);
  /* d'autres choses */

  return 0;
}
```

Tableaux

Exercices

Exercice 1 : Faire une fonction qui remplisse le triangle de Pascal dans un tableau 2D

Exercice 2 : Jeu des allumettes : A partir de la situation initiale, chacun son tour, un joueur retire entre une et trois allumettes sur une ligne. Celui qui retire la dernière a perdu.

X
XXX
XXXXX
XXXXXXX

Exercice 3 : Jeu de la vie. Expliquer à l'oral.

AVANCEMENT

8 HEURES

Allocation dynamique

Introduction

Dans le cas de tableaux, la taille du tableau est fixée à la compilation. Parfois, l'espace mémoire nécessaire ne peut être fixé qu'à l'exécution (ouverture d'une image, travail sur des fichiers de taille donnée...).

On dispose alors de fonctions permettant de réserver un certain espace mémoire spécialement dédié à notre programme.

La fonction va RESERVER un certain nombre d'octets consécutifs et vous renvoyer l'adresse du premier sous forme de pointeur.

```
void *malloc (size_t nb_octets);
```

En cas de problème, la fonction renvoie la valeur 0 (pointeur NULL)

Allocation dynamique

Premières manipulations

Admettons que l'on veuille faire un programme qui affiche les N premiers entiers qui sont des nombres premiers. N est fixé par l'utilisateur.

Comment faire ?

Nous allons demander N à l'utilisateur, réserver de la mémoire pour pouvoir y mettre nos N nombres. Nous allons ensuite faire défiler les nombres entiers pour remplir le tableau jusqu'à obtention de N nombres premiers.

Une solution pratique serait d'avoir une fonction qui nous dise si un nombre est premier ou pas.

Allocation dynamique

Exercice

Exercice : Réaliser le programme précédent.

Le squelette du programme sera :

```
int est_premier(int val)
{
  int premier=0;

  /* si val est premier,
     premier vaudra 1,
    premier vaudra 0 sinon */

  return premier;
}
```

```
int main(void)
{
  int *tab_premiers;
  int i, test_fin=0;

  /* Demander N */
  /* Allouer espace memoire */

 /* Remplir le tableau */

 /* Afficher le tableau */

 /* liberer memoire allouee */
 return 0;
}
```

Allocation dynamique

Autres fonctions et remarques

Les fonctions suivantes pourront vous être utiles :

```
int sizeof (type);                          /* donne le nombre d'octets employé par le type 'type' */

void * malloc (size_t taille);              /* reserve 'taille' octets */
void * calloc (size_t nb_elt, size_t taille_elt);   /* reserve 'nb_elt' cases
                                                       de taille_elt' octets chacune */
void * realloc(void*ptr, size_t size);      /* re-alloue un espace...*/

void free (void* ptr);                      /* libère l'espace mémoire
                                                commencant à l'adresse ptr */
```

Toutes les fonctions alloc renvoient l'adresse du pointeur obtenu et NULL en cas d'échec

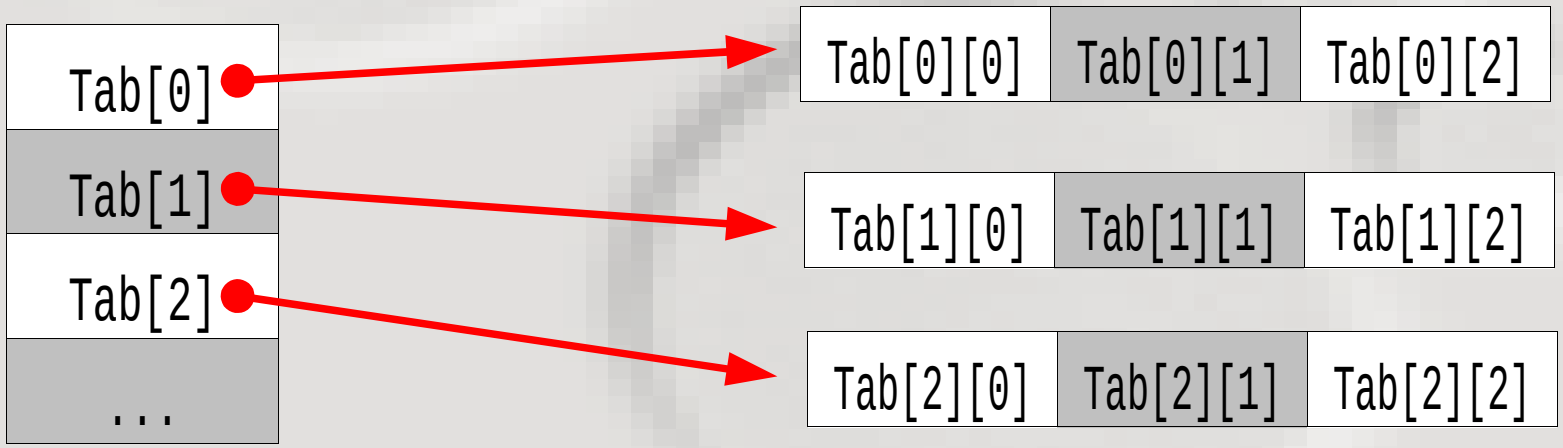
Que contient le tableau après allocation ? Des zéros.

Allocation dynamique

Doubles pointeurs.

Les tableaux bidimensionnels sont en fait des tableaux monodimensionnels un peu particuliers (les lignes sont collées les unes à la suite des autres)

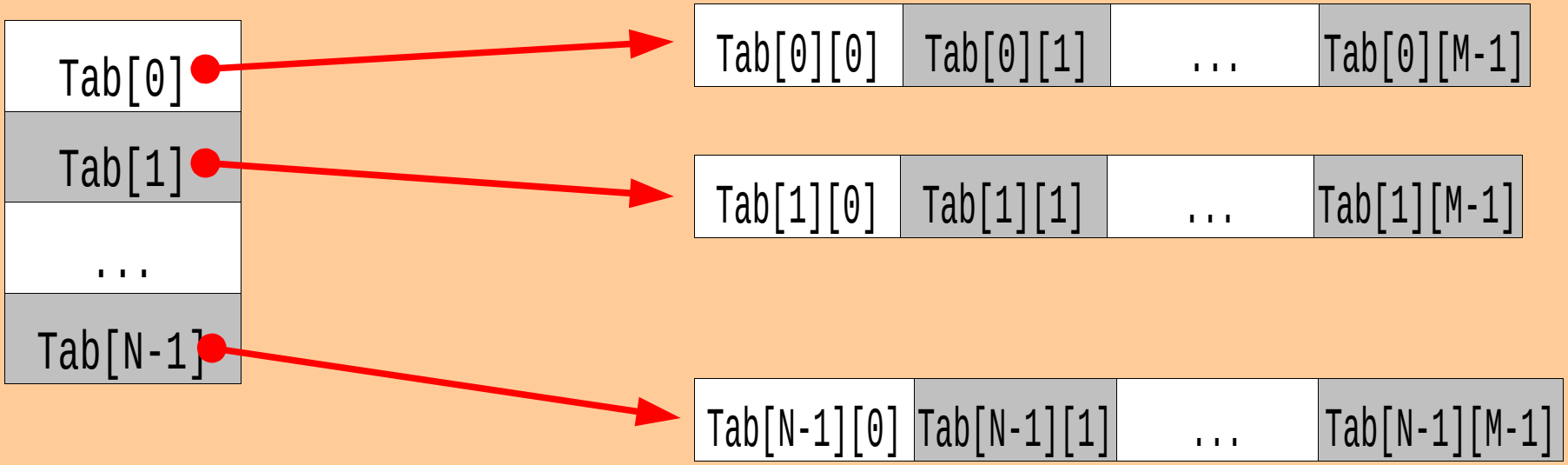
Problème : Un bloc de mémoire consécutif de N*M cases peut être difficile à trouver. Les doubles pointeurs vont nous permettre de séparer les espaces mémoires de chaque ligne.



Allocation dynamique

Doubles pointeurs.

Faire l'allocation correspondant au tableau suivant :



Recenser les différentes façons de nommer une case dans ce tableau.

Allocation dynamique

Doubles pointeurs.

Exercice : Comment libérer (désallouer) un tel tableau ?

Exercice : Refaire le programme du triangle de pascal en n'utilisant que l'espace mémoire strictement nécessaire.

Exercice : Refaire le programme du Jeu de la Vie avec des doubles pointeurs.

AVANCEMENT

10 HEURES

Caractères

Introduction

Comment coder du texte sur un ordinateur ?

Si on admet qu'on peut coder des entiers sous forme de 0 et de 1 (langage binaire), il suffit de définir un code qui a un nombre fait correspondre un caractère.

Pour manipuler des caractères, il suffira ensuite de manipuler des nombres, ce qu'un ordinateur sait faire.

Ce code (correspondance nombre/'lettre') est appelé code ascii.

Dans ce code, on dispose de 256 caractères.

Caractères

Code Ascii

Comment l'afficher :

```
#include <stdio.h>

int main(void)
{
  char i;

  for (i=0;i<256;i++)
    printf("%d = %c\n",i,i);   /* On affiche simplement i sous deux formes...)

  return 0;
}
```

Attention, choses étranges au début : caractères de contrôle (\n, \b...)

Caractères

Remarques sur le code Ascii

- Les chiffres se suivent.
- Les lettres se suivent.
- Les lettres minuscules et majuscules sont différentes.

Admettons qu'on demande un caractère à l'utilisateur avec la commande

```
c = getchar();
```

- Différents tests :
  - L'utilisateur a tapé sur 9

```
if (c == '9')
```

  - L'utilisateur a tapé sur un chiffre

```
if (c >= '0' && c <='9')
```

  - L'utilisateur a tapé sur Enter

```
if (c == '\n')
```

Caractères

Exercice

Exercice : Faire une fonction qui, si le caractère est une lettre en minuscule, renvoie le même caractère en majuscule.

Chaînes de caractères

Introduction

Une phrase est simplement une suite de caractères consécutifs.
On l'appelle chaîne de caractères.

La solution naturelle pour stocker des entités de même type :
un tableau.

Problème : comment indiquer le nombre de caractères de la phrase ?
(le C ne gère pas les questions de taille de tableaux...)

Il faut transporter cette information dans le tableau.

Solution retenue en C : un caractère particulier indique la fin de chaîne :
le caractère '\0' dont le code ascii vaut 0

Chaînes de caractères

Manipulations bas niveau

Exemple de chaine de caractères : `char chaine[256]="chaben";`

Valeur	C	H	A	B	E	N	\0
Adresse	Chaine				Chaine+4		

- Modifier un char dans une chaine : `chaine[4]='i';`
- Ajouter un char dans une chaine : `chaine[6]='e'; chaine[7]=0;`
- Afficher une chaine : `for(i=0;chaine[i]!=0;i++) printf("%d"chaine[i]);`
- Lire une chaine : `for(i=0;(c=getchar())!='\n';i++) chaine[i]=c;`

Chaînes de caractères

Fonctions utiles

Saisie :

- scanf
- fgets

```
char  chaine[256]="chaben";
```

Création :

- sprintf

Affichage :

- printf

Divers :

- strcmp
- strlen
- strcpy

Chaînes de caractères

Exercice

Exercice : Faire un programme qui demande à l'utilisateur une chaîne et lui dise si il s'agit d'un palindrome.

On pourra faire une fonction qui travaille sur un mot sans espace.

Comme certaines phrases sont des palindromes, on pourra tester des phrases palindromes en supprimant les espaces de la phrase et en considérant la chaîne ainsi formée comme un mot.

Intermède et remarques

Règles de programmation

Ces rêgles ont pour but d'améliorer votre programmation dans le sens suivant : faciliter le travail de groupe et la maintenance.

- Noms de variables explicites (favorise la maintenance)
- Une action = une fonction sauf si considérations de temps de calcul.
- Pas de variables globales sauf cas particuliers (force votre attention)
- Méthode (Merise ?)
- Commentez

  (noms de variables, format des données, action des boucles...)

AVANCEMENT

12 HEURES