Table des matières

A. Structure

1. Qu'est ce qu'une structure ?

La structure en programmation est un type de données qui permet de regrouper en une seule unité plusieurs variables éventuellement de types différents. C'est UNE variable qui est un ensemble de variables. C'est très utile pour composer des objets complexes qui réunissent différentes facettes. Par exemple si l'on veut faire un carnet de rendez-vous, chaque rendez-vous suppose des informations du genre :

- un libellé
- un lieu
- une date
- un horaire de début
- un horaire de fin
- une catégorie (bureau, perso, loisir etc.)
- etc.

Toutes ces informations peuvent être regroupées au sein d'une structure et plusieurs structures permettront d'avoir plusieurs rendez-vous. L'intérêt ensuite est de pouvoir avoir un nombre illimité de rendez-vous.

Autre exemple dans un jeu vidéo en 2D un ennemi est définie par une position, un déplacement, un type (rampant, grouillant, serpentant, plombant, assommant), un taux d'agressivité, une couleur, une ou plusieurs images etc. Pour chacune de ces informations il s'agit premièrement de choisir un type de variable approprié et ensuite de regrouper toutes ces informations au sein d'une structure "ennemi" afin de pouvoir avoir un ou plusieurs ennemis dans le jeu.

Chaque élément de la structure est appelé un "champ". Notre structure rendez-vous a au moins 6 champs et la structure ennemi au moins 8 champs (position et déplacement horizontaux et verticaux, type, …).

2. Avoir une structure dans un programme

a. Définir un type de structure

A la base pour définir une structure c'est le mot clé struct suivi d'une liste de déclarations de variables (sans initialisation) entre accolade et d'un point virgule :

```
struct {
  float x,y;      // attention pas possible d'initialiser ses
  float dx,dy;    // variables lors de la définition de la
  int color       // structure
  char lettre;
};
```

Tel quel c'est bien un type de structure mais comme il n'a pas de nom il n'est pas possible de l'utiliser dans tout le programme. Pour nommer son type de structure c'est simple il suffit d'ajouter le nom voulu après le mot clé struct :

```
struct entite {        // nommer son type de structure
  float x,y;
  float dx,dy;
  int color
  char lettre;
};
```

Nous venons de définir le type de structure "struct entite". Attention c'est un nom de type, ça ne correspond pas à une variable mais à un type de variable.

b. Déclarer ses variables structure

Après avoir défini un type de structure, pour avoir des variables ce type c'est comme pour les variables ordinaires. Il faut indiquer le type, donner un nom pour la variable et clore avec un point virgule.

Pour avoir des variables du type struct entite défini ci-dessus nous écrirons quelque part dans le programme :

```
struct entite e1, e2;
```

e1 et e2 sont deux variables du type struct entite.

Remarque

Le nom du type de la structure est indépendant de celui de la variable de ce type et il est possible d'avoir une variable du même nom :

```
struct entite entite;        // fonctionne
```

Par ailleurs il est possible de combiner une définition de type de structure et des déclarations de variables de ce type de structure. La définition de la structure pix ci-dessous peut être suivie d'une suite de déclarations de variables :

```
   struct pix{
          int x, y;
          int color;
   }P1, P2, toto;
```

La définition de la structure struct pix est suivie des déclarations des variables P1, P2 et toto qui sont trois structures de type struct pix.

3. Utiliser une structure

a. Accès aux éléments avec l'opérateur point

Toutes les variables d'une structure, c'est-à-dire tous les champs de la structure sont accessibles via l'opérateur " **.** " dit "point" de la façon suivante :

```
   struct pix  p;

          p.x = 0;
          p.y = 50;
          p.color = 255;
```

La première ligne déclare une structure du type struct pix nommée p. Les trois lignes suivantes initialisent chacun des champs de la variable p. Le champs x prend la valeur 0, le champ y prend la valeur 50 et le champ color la valeur 255.

b. priorité de l'opérateur point

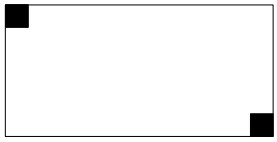
Priorité maximum, tous les autres opérateurs passent après dans l'évaluation d'une expression (Annexe 1 : Priorité et associativité des opérateurs).

c. Une structure comme champ dans une structure

L'imbrication de structures est possible. Une structure peut avoir une autre structure comme champ à condition que son type soit connu au moment de la définition de la structure. Par exemple :

```
struct rgb{      // valeur de rouge, de vert et de bleu
          int r,g,b;    // pour avoir une couleur
};

struct coord{
          int x, y;     // pour stocker une position en 2D
};

struct rectangle{
          struct coord hg;          // coord du coin haut gauche
          struct coord bd;          // coord du coin bas droite
          struct rgb  color;        // couleur du rect
};
```

La structure rectangle permet de stocker les informations de rectangles colorés. La structure rgb concentre les informations de couleurs avec trois champs, un entier pour le rouge, un entier pour le vert et un entier pour le bleu (En RGB _ red, green, blue _ une couleur est obtenue par un mélange de rouge de vert et de bleu). La structure coord permet de stocker les coordonnées d'un point dans un espace à deux dimensions. Pour dessiner un rectangle nous avons besoin de deux points, le point en haut et à gauche et le point en bas et à droite :



La structure finale pour stocker les informations d'un rectangle comprend une structure rgb pour sa couleur et deux structures coord pour chacun des points haut gauche (hg) et bas droite (bd).

Dans cet exemple les structures rgb et coord doivent absolument être définies avant la structure rectangle pour y être connues au moment de sa définition. Dans le cas contraire il y aura une erreur à la compilation.

L'accès aux champs d'une structure se fait avec l'opérateur **.** (point) et rien ne change si le champ est lui-même une structure. L'opérateur point sera utilisé une nouvelle fois pour accéder aux champs de la structure élément de la façon suivante :

```
struct rectangle r1;
          r1.hg.x = 0;
          r1.hg.y = 0;
          r1.bd.x = 100;
          r1.bd.y = 50;
          r1.color.r = 255;
          r1.color.g = r1.color.b = 0;
```

Le type de l'expression (r1.hg) par exemple est une structure du type struct coord et on accède à chacun de ses champs en utilisant une nouvelle fois l'opérateur point, soit les expressions (e1.hg).x et (e1.hg).y et les parenthèses sont inutiles.

d. Initialiser une structure à la déclaration

Une structure peut être initialisée à la déclaration en lui affectant une suite de valeurs entre accolade close par un point virgule. Chaque valeur est affectée à un champ correspondant dans l'ordre et doit en respecter le type. Soit la structure test :

```
struct test {
          float x, y;
          int px, py;
          char lettre;
};
```

et une déclaration suivie d'une initialisation dans le programme :

```
struct test t1={ 1.5, 2.0, 150,300,'A'};
```

Pour la variable t1 de type struct test, 1.5 est affecté au champ x, 2.0 est affecté au champ y, 150 est affecté au champ px, 300 au champ py et le champ lettre prend la valeur 'A'.

Dans le cas de structures imbriquées il y a le choix entre une seule liste pour les valeurs affectées ou le détail entre accolades pour chaque structure, par exemple :

```
struct date{
          int jour, mois, annee;
};
struct time{
          int heure, minute, seconde;
};
struct moment{
          struct date date;
          struct time time;
};
```

Dans le programme, un exemple de déclaration suivie d'une initialisation :

```
   struct moment m1={ 25,12,2006,13,45,30};
```

Les champs de m1 valent respectivement 25 pour jour, 12 pour mois, 2006 pour annee, 13 pour heure, 45 pour minute et 30 pour seconde.

On peut également détailler le contenu de chaque structure imbriquée de la façon suivante :

```
struct moment m1={  {25,12,2006},
                                          {13,45,30}      };
```

Si le nombre des valeurs de la liste est supérieur au nombre des champs il y a une erreur de compilation. En revanche si le nombre des valeurs de la liste est inférieur, les champs restants sont initialisés à 0. Ce test permet de le vérifier :

```
int main()
{
struct moment m={{1}, {3,4}};

   printf("m.date.jour=%d\n",m.date.jour);
   printf("m.date.mois=%d\n",m.date.mois);
   printf("m.date.annee=%d\n",m.date.annee);
   printf("m.time.heure=%d\n",m.time.heure);
   printf("m.time.minute=%d\n",m.time.minute);
   printf("m.time.seconde=%d\n",m.time.seconde);
          return 0;
}
```

Résultat :

```
m.date.jour=1
m.date.mois=0
m.date.annee=0
m.time.heure=3
m.time.minute=4
m.time.seconde=0
```

e. Copier une structure

Les copies et les affectations de structure à structure de même type sont autorisées, par exemple :

```
   struct coord pt1, pt2;
          pt1.x=100;
          pt1.y=200;

          pt2=pt1;
```

La structure pt1 est initialisée avec les valeurs 100 et 200, ensuite la structure pt1 est affectée à la structure de même type pt2. Les valeurs de pt1 sont recopiées dans pt2 (Les structures sont dites des lvalues _leftvalue_).

4. Mise en pratique : définir, déclarer, initialiser des structures

Exercice 1
Une menuiserie industrielle gère un stock de panneaux de bois. Chaque panneau possède une largeur, une longueur et une épaisseur en millimètres ainsi que le type de bois. Il y a trois types de bois : pin (code 0), chêne (code 1), hêtre (code 2).

- □ Définir une structure panneau contenant toutes les informations relatives à un panneau de bois.
- □ Dans un programme initialisez deux panneaux à la déclaration et deux panneaux avec des valeurs aléatoires. Affichez les valeurs.

Exercice 2
Un grossiste de composants électronique vend quatre types de produits :

- □ des cartes mères (code 1)
- □ des processeurs (code 2)
- □ des barrettes mémoire (code 3)
- □ des cartes graphiques (code 4)

Chaque produit possède une référence, qui est un nombre entier, un prix en euros et une quantité disponible. Définir une structure produit qui code un produit et dans un programme initialisez deux produits à la déclaration et deux produits avec des valeurs entrées par l'utilisateur. Affichez les valeurs et clonez le produit le plus cher.

Exercice 3
Soit un programme de flocons de neige qui tombe du haut de la fenêtre console en bas. Chaque flocon est une lettre. Donnez une structure de données qui permet de coder un flocon. Dans un programme déclarez quatre flocons. Initialisez un flocon à la déclaration et un autre avec des valeurs aléatoires. Copiez chaque flocon et affichez les valeurs des quatre flocons de façon claire.

Exercice 4
Le monde où se trouve le player est parcouru par des ennemis. Donnez les caractéristiques essentielles d'un ennemi que vous imaginez et la structure de données le mieux adaptée. Initialisez un ennemi à la déclaration et un autre avec des valeurs aléatoires. Affichez les valeurs de votre ennemi. Faites un clone.

B. Structures et fonctions

1. Retourner une structure

Comme la copie de structure est autorisée, les structures peuvent être prises comme valeurs de retour. Soit la structure coord :

```
struct coord{
   int x, y;
};
```

Voici typiquement une fonction qui permet d'initialiser une structure après sa déclaration n'importe où dans le programme :

```
   struct coord init_point1()
   {
   struct coord t0;
          t0.x = rand()%1024;
          t0.y = rand()%768;
          return t0;
   }
```

Dans la fonction :

- □ Une structure du type souhaité est déclarée
- □ Chacun de ses champs est initialisé avec des valeurs aléatoires ou calculées
- □ Pour finir la structure résultante est retournée au contexte d'appel

Bien sur des valeurs peuvent être transmises en paramètre, ça donne par exemple :

```
struct coord init_point2(int x, int y)
{
struct coord t0;
   t0.x = x;
   t0.y = y;
   return t0;
}
```

Et dans le programme la structure retournée par la fonction est affectée à une structure du contexte d'appel :

```
int main()
{
struct coord pt1, pt2;

   pt1 = init_point1();
   pt2 = init_point2(40, 40);
   printf("pt1.x=%d, pt1.y=%d\n"
          "pt2.x=%d, pt2.y=%d\n",pt1.x,pt1.y,pt2.x, pt2.y);
   return 0;
}
```

L'appel de init_point1() renvoie une structure coord initialisée avec des valeurs aléatoires et elle est affectée à la variable coord pt1. L'appel de init_point2() renvoie une autre structure coord initialisée avec (40,40) et affectée à la variable coord pt2.

2. Structures en paramètre de fonction

Une structure est passée à un paramètre de fonction de même type par valeur comme n'importe quel autre type de variable. La structure passée est copiée dans le paramètre de la fonction.

Attention !

Soit par exemple la fonction ci-dessous elle permet de faire progresser de 1 la position horizontale d'un point. Si la position dépasse 1000 elle retourne à 0 :

```
void modif(struct coord pt)
{
   pt.x++;
   if (pt.x>1000)
       pt.x=0;

}
```

Dans un programme l'appel est le suivant :

```
int main()
{
struct coord p={ 10,20};
   printf("p.x=%d, p.y=%d\n", p.x, p.y);
   modif(p);
   printf("p.x=%d, p.y=%d\n", p.x, p.y);
   return 0;
}
```

Qu'imprime le programme ?

```
p.x=10, p.y=20
p.x=10, p.y=20
```

Il n'y a pas de modification de la structure p. La structure en paramètre de la fonction modif() est locale à la fonction. C'est une autre variable. Au moment de l'appel l'affectation implicite suivante est réalisée :

```
modif(pt = p);
```

Mais à l'issue de la fonction la structure p dans le contexte d'appel n'a pas été touchée.

Pour avoir une fonction de modification des valeurs contenues dans une structure il faut passer la structure à modifier en argument et retourner la structure modifiée, ce qui donne :

```
struct coord  modif(struct coord pt)
{
   pt.x++;
   if (pt.x>1000)
       pt.x=0;
   return pt;
}
```

avec l'appel :

```
int main()
{
struct coord p={10,20};;
   printf("p.x=%d, p.y=%d\n", p.x, p.y);
   p=modif(p);
   printf("p.x=%d, p.y=%d\n", p.x, p.y);
   return 0;
}
```

Cette fois la copie modifiée dans la fonction est affectée à la structure p locale au main() et le programme imprime :

```
p.x=10, p.y=20
p.x=11, p.y=20
```

3. Expérimentation : une entité mobile à l'écran

```
/*************************************************************
   L'obectif est de faire UNE entité qui se déplace à l'écran.
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include <windows.h>

// en global définition et déclaration du type de ma structure.
struct entite{
   float x,y;     // position de l'entité
   float dx,dy;   // sa valeur de déplacement
   int color;     // sa couleur
   char lettre;   // son apparence
};

// constantes globales pour les limites de la zone de jeu
const int TX = 40;
const int TY = 24;
```

```
// mes déclarations de fonctions
struct entite  init           (void);
void           affiche        (struct entite e,int color);
struct entite  move           (struct entite e);
int            top            (int*start,int dur);

/
*****************************************************************
TEST 1 : UNE SEULE ENTITE
*****************************************************************/
int main()
{
struct entite e;
int start=0;

   // initialisation générateur aléatoire
   srand(time(NULL));

   // initialisation structure
   e=init();

   // boucle du jeu
   while (!kbhit()){

     // l'action est exécutée une fois toutes les 75 millisecondes
      if (top(&start,75 )){

         // effacer à la position courante
         affiche(e,0);

         // avancer
         e=move(e);

         // affichage
         affiche(e,e.color);
      }
   }
   return 0;
}
/
*****************************************************************
En valeur de retour la structure se comporte comme une variable
ordinaire
*****************************************************************/
struct entite init()
{
struct entite e;
   e.x=rand()%TX;
   e.y=rand()%TY;
   e.dx = ((float)rand()/RAND_MAX)*4 - 2; // val float entre
                                          // -2 et 2
   e.dy = ((float)rand()/RAND_MAX)*4 - 2;
   e.color= rand()%256;
   e.lettre='A'+rand()%26;
   return e;
}
/
*****************************************************************
```

```
En paramètre de fonction la structure se comporte comme une
variable ordinaire. Il s'agit d'un passage par valeur.
*****************************************************************/
void affiche(struct entite e,int color)
{
   gotoxy(e.x,e.y);
   textcolor(color);
   putchar(e.lettre);
}
/
*****************************************************************
*****************************************************************/
struct entite move (struct entite e)
{
   // avance
   e.x+=e.dx;
   e.y+=e.dy;

   // contrôle bords
   if( e.x<0 || e.x>TX)
      e.dx*=-1;
   if( e.y<0 || e.y>TY)
      e.dy*=-1;

   return e;
}
/
*****************************************************************
*****************************************************************/
int top(int*start,int dur)
{
int res=0;
   if (clock()> *start+dur){
      res=1;
      *start=clock();
   }
   return res;
}
```

4. Mise en pratique 13 : structures et fonctions

Exercice 1

Une menuiserie industrielle gère un stock de panneaux de bois. Chaque panneau possède une largeur, une longueur, une épaisseur en millimètres, un volume et un type de bois. Il y a trois types de bois : pin (code 0), chêne (code 1), hêtre (code 2).

- Définir une structure panneau contenant toutes les informations relatives à un panneau de bois.
- Écrire les fonctions de saisie d'un panneau de bois.
- Écrire une fonction d'affichage d'un panneau de bois
- Écrire une fonction qui calcule le volume en mètre cube d'un panneau.
- Écrire une fonction qui affiche le volume d'un panneau de bois

Exercice 2

Un grossiste de composants électroniques vend quatre types de produits :

- des cartes mères (code 1)
- des processeurs (code 2)

- □ des barrettes mémoire (code 3)
- □ des cartes graphiques (code 4)

Chaque produit possède une référence, qui est un nombre entier, un prix en euros et une quantité disponible.

- □ Définir une structure "produit" qui code un produit
- □ Écrire une fonction de saisie des données d'un produit
- □ Écrire une fonction d'affichage des données d'un produit
- □ Écrire une fonction qui permet à un utilisateur de saisir la commande d'un produit.

L'utilisateur identifie le produit et saisit la quantité demandée. L'ordinateur affiche toutes les données de la commande y compris le prix.

Exercice 4
Soit les deux types de structure "date" et "personne" :

```
struct date{
          int jour, mois, annee;
};
struct personne{
          int identifiant;
          struct date date_embauche;
          struct date date_poste;
};
```

- □ Ecrire une fonction pour initialiser une structure de type "personne".
- □ Ecrire une fonction d'affichage de façon à obtenir soit l'un soit l'autre des deux affichages ci-dessous :

```
Identifiant : 1224
date embauche (jj mm aa) : 16 01 08
date poste = date embauche ? (O/N)  : O

Identifiant : 1224
date embauche (jj mm aa) : 16 01 08
date poste = date embauche ? (O/N)  : N
date poste (jj mm aa) :  01 09 08
```

Exercice 5
Soit une entité dans un jeu vidéo en mode console. Elle est définie par une position, un déplacement, un type (rampant, grouillant, serpentant, plombant, assommant), une couleur. L'entité a également un nom (une lettre) qui sert pour son apparence et une série de taux : taux d'agressivité, de colère, de convoitise, de faim, de peur.
Définir la structure de données pour coder une entité. Faire une fonction d'initialisation, une fonction de mise à 0 (reset) et une fonction d'affichage. Tester dans un programme avec un menu : quitter, afficher, initialiser, reset.

C. Typedef, enum et #define

1. Utiliser un typedef

Le C donne la possibilité de définir ces propres types à partir des types de base. Pour ce faire il faut utiliser la commande typedef. Le principe est simple : déclarer une variable du type que vous voulez, ajoutez typedef devant ...et le nom de la variable devient synonyme du type de cette variable. Par exemple :

```
typedef int toto;
```

Permet de définir le type "toto" dans un programme ; toto devient synonyme de int et peut être utilisé à la place dans le programme :

```
typedef int toto;

int main()
{
toto test=80;
          printf("la variable test vaut : %d\n",test);
          return 0;
}
```

L'intérêt de pouvoir rebaptiser ses types dépend de certaines situations, par exemple créer un langage en français. Mais c'est utilisé quasi systématiquement avec les structures afin de faire disparaître le mot clé struct.

Soit par exemple le type struct ennemi dans un programme :

```
struct ennemi{
   char type;          // type de l'ennemi (une lettre)
   float nuisance;     // potentielle de nuisance
   float x,y,px,py;    // position et déplacement
   int force;          // indicateur de force pour les combats
};
```

En ajoutant le mot clé typedef devant on peut définir un synonyme de struct ennemi de la façon suivante :

```
typedef struct ennemi{
   char type;
   float nuisance;
   float x,y,px,py;
   int force;
}t_ennemi;
```

et de façon plus concise nous pouvons définir directement notre type de structure :

```
typedef struct{
   char type;
   float nuisance;
   float x,y,px,py;
   int force;
}t_ennemi;
```

t_ennemi devient nom de type pour la structure dans les deux cas du fait de l'utilisation de typedef et t_ennemi peut être utilisé partout dans le programme :

```
int main()
{
t_ennemi E;
int i;
   srand(time(NULL));
   E=init_ennemi();
   affiche_ennemi(E);
```

```
   return 0;
}
```

Le préfixe t_ est pratique. Il permet de préciser qu'il s'agit d'un type et non d'un nom de variable. Ça n'a rien d'obligatoire.

2. Utiliser un enum

Le mot clé enum, pour énumération, permet de définir une suite de valeurs entières constantes identifiées par des noms. La numérotation est faite automatiquement. Par défaut elle commence à 0 pour le premier nom et s'incrémente de un à chaque nom suivant, par exemple :

```
enum { NON, OUI, PEUT_ETRE };
```

Pour cette enum NON vaut 0, OUI vaut 1 et PEUT_ETRE vaut 2.

Une valeur peut être spécifiée pour un élément de la liste, cet élément prend alors cette valeur et les suivants sont incrémentés de un en un à partir de cette valeur par exemple :

```
enum { start=77, run, accelerer, slow=-20, stop, inverse};
```

start vaut 77, run vaut 78, accelerer vaut 79, slow vaut -20 et stop vaut -19, inverse vaut -18.

Eventuellement l'enum peut porter un nom et il est alors possible de l'utiliser comme type pour déclarer des variables entières. L'intérêt est sémantique et méthodologique. Par exemple dans l'exemple ci-dessous nous supposons que t1 prendra des valeurs correspondant à celle de l'enum test :

```
   enum test { start, run, accelerer, slow=100, inverse, stop};

int main()
{
enum test t1;
int i;
   for (i=0; i<=stop; i++){
       t1=rand();
       switch(t1){
           case start :   /*instructions*/   break;
           case run :     /*instructions*/   break;
               (...)
           case stop :    /*instructions*/   break;
           default :      /*instructions*/   break;
       }
   }
   return 0;
}
```

Mais en C il n'y a pas de contrôle sur la valeur de t1 qui se comporte comme un int ordinaire. L'utilisation la plus intéressante et la plus importante de l'enum est de pouvoir organiser des ensembles de constantes entières reliées entre elles méthodiquement pour faire sens. Par exemple dans un jeu :

```
typedef enum{ NORD, EST, SUD, OUEST} direction;
```

permet de définir quatre constantes entières pour identifier les directions que peut prendre un personnage. Direction est une variable du type de l'enum et avec le typedef devient synonyme de cet enum. Nous pouvons ensuite utiliser direction comme nom de type pour l'enum :

```
int main()
{
direction d;
   srand(time(NULL));
   d=rand()%(OUEST+1);
   printf("%d\n", d);
   return 0;
}
```

3. Utiliser un #define

Dans le langage C #define est ce que l'on appelle une "macro", une directive préprocesseur. #include, #if, #ifndef, #ifdef, #else, #endif, #undef sont d'autres directives. Elles sont introduites par # au début d'une ligne et se terminent en fin de ligne.

Du point de vue de la compilation les directives sont liées à un prétraitement par le préprocesseur dont l'objet est de préparer le code source pour le compilateur. C'est une première étape de la compilation. Elle fournit des portions de code au compilateur et également supprime tous les commentaires du code source du programme.

Le mot clé define définit un nom symbolique qui représente soit une valeur constante, soit une fonction, mais en fait n'importe quel texte de remplacement. On a une écriture du genre :

```
   #define nom         texte de remplacement
```

Et dans le programme à chaque fois que l'on appelle **nom** c'est *texte de remplacement* qui est substitué. La syntaxe de *texte de remplacement* est en fait indépendante du reste du langage ce qui permet des utilisations variées parfois astucieuses et surprenantes. Toutefois après remplacement du symbole **nom** par *texte de remplacement* dans le programme, le code résultant doit être correct syntaxiquement. Si *texte de remplacement* contient du code incomplet, il devra avoir été complété autour de **nom**, au moment de l'appel de **nom**.

Au besoin, un antislash \ , ajouté comme dernier caractère de la ligne indique que la ligne se poursuit avec la ligne suivante.

Typiquement le #define est utilisé en C pour identifier des valeurs constantes. Par exemple ici une taille de zone de jeu en mode console :

```
#define TX      80
#define TY      24
```

Cette expression signifie que TX est équivalent à 80, TY à 24. Dés lors TX et TY peuvent être utilisés partout dans le programme à la place des valeurs effectives. Par exemple dans des boucles :

```
int main()
{
```

```
   int x, y;
   for (y=0; y<TY; y++)
       for (x=0; x<TX; x++)
           printf("%c", 'A'+((y*TY+x)%26));
   return 0;
}
```

L'intérêt est qu'il est très facile de modifier le programme ensuite pour qu'il tourne avec d'autres valeurs. Il suffit de modifier la valeur de remplacement du #define et il n'y a pas à toucher au code, par exemple dans le programme précédent si j'écris :

```
#define TX      40
#define TY      12
```

et que je recompile, le programme fonctionne désormais avec les valeurs 40 et 12.

Remarque :

L'utilisation des #define en C est aujourd'hui un peu contreversée. Brian Kernighan lui-même, créateur du langage C avec Denis Ritchie, déclare "le préprocesseur C est un outil puissant mais néanmoins un peu dépassé, et les macros constituent une méthode de programmation dangereuse car elles modifient la structure lexicale du programme de manière sous-jacente." (KERNIGHAN, "la programmation en pratique", p.23]).

4. Mise en pratique : typedef, enum, #define

Exercice 1
Définir un type pour la structure : struct pix { int x; int y; int color };
faire une fonction qui bombarde de pix la fenêtre console.

Exercice 2
Modifier la DEMO13 (chapitre 13 : (une entité qui se déplace à l'écran) de façon à pouvoir déplacer l'entité avec les flèches du clavier. Vous définirez un enum pour les directions (NORD, EST, SUD, OUEST) un type pour la structure et deux define pour la taille de la zone de jeu.

Exercice 3
Faire un joueur de foot qui joue seul avec un ballon.

D. Tableaux statiques

1. Qu'est ce qu'un tableau ?

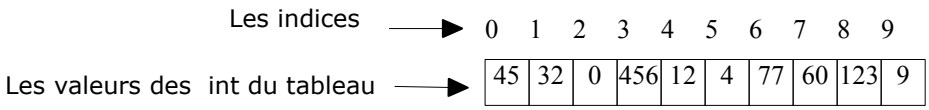
Un tableau est un ensemble d'objets de même type : un tableau de int, de float de double, de structure de même type etc.

Le fait qu'il soit statique signifie qu'il est non dynamique c'est à dire que sa taille est fixe et que son espace mémoire est alloué par la machine à la déclaration.

Chaque élément du tableau est numéroté du premier 0 au dernier qui est nombre d'élément dans le tableau moins un. Le numéro d'un élément est appelé son indice.

L'indice, associé à l'opérateur crochet [ ] va permettre d'accéder à l'élément correspondant.

Par exemple dans un programme avoir un tableau de 10 int, c'est avoir un regroupement de 10 int, en une seule variable de type tableau de int. Les indices des éléments vont de 0 pour le premier à 9 pour le dernier. Chaque int du tableau a sa propre valeur. Nous pouvons le représenter de la façon suivante :



l'entier n°4 vaut 12, l'entier n°1 vaut 32, l'entier n°8 vaut 123, l'entier n°9 vaut 9 etc.

2. Avoir un tableau statique dans un programme [•] [▼][▲]

a. Définir et déclarer un tableau

Pour définir et déclarer un tableau dans un programme il faut donner :

- le type des éléments
- un nom pour le tableau
- le nombre des éléments entre crochet
- un point virgule

Soit le formalisme :

```
<type> <nom> <[ constante entière] > < ; >
```

Par exemple :

```
int tab [10];          // déclaration d'un tableau de 10 int
float f[90]            // déclaration d'un tableau de 90 float
```

La déclaration peut se faire dans n'importe quel bloc d'instruction du programme et les règles de visibilité sont les mêmes que pour les variables simples.

Attention !

Le nombre des éléments doit toujours être une valeur entière et constante. Ça ne peut pas être une variable. Pour avoir un nombre variable d'éléments il faut un tableau dynamique c'est à dire utiliser des pointeurs (voir le chapitre pointeurs).

b. Utiliser des #define pour les tailles

Pour la taille du tableau il est préférable d'utiliser une macro-constante #define ou une constante d'enum par exemple :

```
#define NBMAX   50

int main()
{
```

```
int test[NBMAX];

   (…)

   return 0;
}
```

En effet beaucoup d'opérations sur les tableau se réfèrent à sa taille notamment les boucles qui permettent de passer en revue chaque élément du tableau. Pour pouvoir tester son programme avec des tailles différentes du tableau il suffit alors de modifier la valeur du #define, de recompiler et lancer son programme. Il n'y a pas pas besoin de retrouver toutes les opérations faites sur le tableau et à chaque fois de modifier la taille.

3. Utiliser un tableau

a. Accès aux éléments du tableau avec l'opérateur crochet [ ]

Pour accéder à un élément du tableau il faut utiliser l'opérateur crochet et l'indice de l'élément visé, par exemple :

```
int toto[5];    // les indices sont 0, 1, 2, 3, 4 (la taille 5
                // n'est pas un indice du tableau

   toto[0] = 50;       // mise à 50 du premier élémnt d'indice 0
   toto[1] = 60;       // mise à 60 du deuxième d'indice 1
   toto[4] = 100;      // mise à 100 du dernier d'indice 4

   toto[2]=toto[0]+toto[1];  // met à 110 l'élément troisième
                             // d'indice 2
```

b. Priorité opérateur crochet

Dans l'exemple ci-dessus chaque élément du tableau est un int et toutes les opérations possibles sur les int sont possibles et c'est vrai quelque soit le type des éléments du tableau. Toutes les opérations possibles sur le type d'élément du tableau sont faisables avec les éléments du tableau.

Cela vient de l'opérateur [ ] qui a la priorité maximum comme le point des structures et les parenthèses de fonctions (Annexe 1 : Priorité et associativité des opérateurs)

c. Débordement de tableau

Attention !

D'une façon générale il est interdit d'aller écrire dans une zone mémoire non réservée. Un débordement de tableau, c'est-à-dire une tentative d'accès mémoire en dehors du tableau à partir du tableau provoque une erreur dans le fonctionnement du programme. Le résultat dépend de la gestion de la mémoire, un bug curieux, la sortie immédiate du programme, un crash, un plantage etc. . Malheureusement les débordements de tableaux ne sont pas repérables à la compilation du programme :

Par exemple :

```
int main()
{
float test[5];

   test[-1] = 9.5;           // provoque bug, erreur ou crash
   test[5] = 10.0;           // provoque bug, erreur ou crash
   return 0 ;
}
```

d. Initialiser un tableau à la déclaration

Un tableau peut être initialisé au moment de sa déclaration avec des valeurs constantes. Voici les différents cas possibles :

```
int tab[7]={456, 76, 82, 12, 4, 3, 0};
char msg[3]={'h','o','l','a','!','!' };    // erreur !
float f[50]={1.5, 1.67, 2.33};             // complété avec des 0
short sh[ ] ={12, 34,5678};                // prend la taille 3
```

Si le nombre des constantes de l'initialisation est supérieur à la taille du tableau il y a une erreur de compilation.

Si le nombre des valeurs est inférieur à la taille du tableau le reste du tableau et complété avec des 0.

Un tableau initialisé lors de sa définition peut ne pas avoir de taille explicite dans ce cas il aura la taille du nombre des éléments fournis. Le tableau sh a une taille de trois.

e. Parcourir un tableau avec une boucle for

Comment initialiser un tableau comprenant plusieurs milliers d'éléments ?

A l'initialisation c'est fastidieux. Par exemple un tableau de 5000 int à mettre tous à 10 :

```
int titi[5000]={10,10,10,10,…(5000 éléments !) …, 10,10};
```

Où pire avec une valeur aléatoire :

```
   titi[0] = rand();
   titi[1] = rand();
   ... 5000 éléments !
```

La boucle for permet facilement de faire défiler tous les indices d'un tableau et ainsi d'accéder à chaque élément quelque soit le nombre :

```
int i;
   for (i=0; i<5000; i++)
       titi[i] = rand();
```

la variable i prend successivement toutes les valeurs de 0 à 4999. Pour chaque valeur i sert d'indice du tableau et pour chaque indice les instructions du bloc sont effectuées. Ici une valeur aléatoire est affectée à l'entier correspondant dans le tableau.

L'utilisation d'un #define pour la taille est préférable, ce qui donne par exemple dans un programme complet :

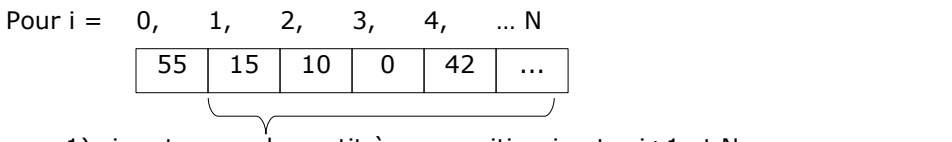
```
#define NBMAX          5000

int main()
{
int tab[NBMAX];
int i;
   for (i=0; i<NBMAX; i++){
       tab[i]=rand()%256;
       printf("tab[%4d]=%4d\n",itab[i]);
   }
   return 0;
}
```

f. Trier un tableau

Le fait d'avoir à organiser ses données selon un ordre ou un autre est fréquent en informatique et il existe de nombreux algorithmes de tris (voir par exemple l'ouvrage de Robert Sedgewick *Algorithmes en langage C*). Voici un algorithme de tri élémentaire, le tri par sélection. Nous allons classer en ordre croissant toutes les valeurs d'un tableau d'entier.

Principe

A partir d'une position i dans le tableau (au départ i = 0) on regarde dans la suite du tableau pour chaque position j (au départ j = i + 1) s'il y a une valeur plus petite. Si oui on permute les valeurs des positions i et j. Ensuite on avance i de 1 et on recommence à partir de la nouvelle position j à j+1 :



1) si on trouve plus petit à une position j entre i+1 et N
2) Permuter les deux valeurs aux indices i et j
3) lorsque j est arrivé à la fin du tableau, avancer i de 1 et recommencer jusqu'à ce que i soit égal à N-1, dernier indice du tableau

Algorithme en C

Pour faire tourner dans un programme l'algorithme ci-dessous il faut pensez à initialiser son tableau d'abord, l'afficher le trier et le réafficher pour constater le résultat.

```
#define N 50

int i,j, tmp, tab[N];

   //… initialiser le tableau avant … puis affichage
   for (i=0; i<N; i++)              // pour chaque i
       for (j=i+1; j<N; j++)        // regarder à partir de i+1
           if (tab[j]<tab[i]){      // si élément plus petit
               tmp=tab[i];          // et si oui permuter
               tab[i]=tab[j];
               tab[j]=tmp;
           }

   // … affichage résultat tableau trié
```

Il y a moyen d'optimiser un peu cette méthode (voir les exercices). Notamment il peut être intéressant d'éviter d'avoir plusieurs permutations sur un même parcours de j ce qui ralentit un peu le traitement.

4. Tableaux à plusieurs dimensions

a. Déclarer un tableau à plusieurs dimensions

Il est possible d'avoir un tableau à plusieurs dimensions. Il suffit de spécifier pour chaque dimension sa taille avec l'opérateur crochet.

Matrice à deux dimensions

Un tableau à deux dimensions est aussi appelé une matrice. Voici par exemple une matrice de float :

```
float mat[3][4]; // est un tableau de float à 2 dimensions ce qui
                 // revient à avoir ici 3 tableaux de 4 float
```

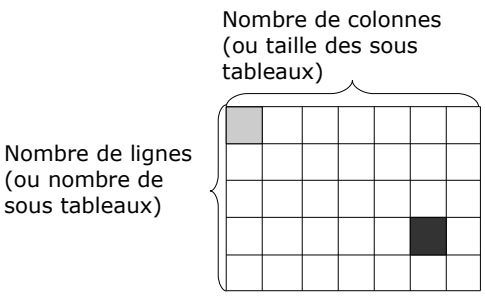
Comme le spécifie Kernighan créateur du langage C "en C un tableau à deux dimensions est en fait un tableau à une dimension dont chaque élément est un tableau". Une matrice c'est un tableau de tableau.

Pour localiser un élément dans un tableau à deux dimensions il faut deux indices. Le premier pour dire dans quelle tableau il se trouve, le second pour indiquer sa position dans ce tableau. Par exemple :

```
mat[1][3]=0;
```

désigne l'élément n°3 dans le tableau n°1.

La première dimension correspond au nombre de lignes, la seconde au nombre de colonnes. Le nombre total d'éléments est lignes*colonnes. L'organisation lignes, colonnes peut se visualiser ainsi :



En noir la position mat[3][5],
En gris la position mat[0][0].

Le parcours des indices de droite, colonnes, est consécutif en mémoire. Pour cette raison c'est le plus rapide. Le parcourt des indices de gauche, lignes, suppose des sauts de la tête de lecture en mémoire. Dans cette exemple des sauts de 7 int en 7 int. Pour cette raison le parcours est moins rapide.

Tableaux à N dimensions

On peut avoir des tableaux à un nombre quelconque de dimensions :

```
char tata[3][3][3];
float titi[5][7][8][23][45][56][123];
etc.
```

Mais il faudra à chaque fois un nombre d'indices correspondant au nombre des dimensions pour localiser un élément du tableau.

Déployer toutes les positions d'un cube de 3*3*3 éléments donne par exemple:

```
tata[0][0][0]   tata[0][0][1]      tata[0][0][2]
tata[0][1][0]   tata[0][1][1]      tata[0][1][2]
tata[0][2][0]   tata[0][2][1]      tata[0][2][2]
tata[1][0][0]   tata[1][0][1]      tata[1][0][2]
tata[1][1][0]   tata[1][1][1]      tata[1][1][2]
tata[1][2][0]   tata[1][2][1]      tata[1][2][2]
tata[2][0][0]   tata[2][0][1]      tata[2][0][2]
tata[2][1][0]   tata[2][1][1]      tata[2][1][2]
tata[2][2][0]   tata[2][2][1]      tata[2][2][2]
```

Pour déployer un tableau à sept dimensions c'est le même principe avec sept indices a, b, c, d, e, f, g qui respectent les tailles maximum de chaque dimension :

```
titi[0] [0] [0] [0] [0] [0] [0] = rand();
titi[0] [0] [0] [0] [0] [0] [1] = rand();
etc.
titi[a] [b] [c] [d] [e] [f] [g] = rand();
```

b. Initialisation à la déclaration

Une matrice peut être initialisée soit de façon continue soit ligne par ligne. Par exemple :

```
int mat[4][2]={1,2,3,0,7,6,5,4};
```

est équivalent à :

```
int mat[4][2]={ {1,2},
                {3,0},
                {7,6},
                {5,4}  };
```

Comme pour les tableaux à une dimension si le nombre des valeurs est inférieur à la taille du tableau, l'initialisation du tableau est complétée par des 0 :

```
      float test[3][3]={ {1,2},
                         {5.666}  };
```

est équivalent à :

```
      float test[3][3]={  {1, 2, 0},
                          {5.666, 0, 0},
                          {0, 0, 0}  };
```

c. Parcourir un tableau à plusieurs dimensions

Pour parcourir un tableau à plusieurs dimensions il faut une boucle par dimensions et toutes les boucles sont imbriquées. Par exemple pour initialiser une matrice il faut :

1) passer en revue chaque ligne et

2) pour chaque ligne parcourir toutes les colonnes

IL y a ainsi deux boucles, une pour les lignes et une pour les colonnes ce qui donne par exemple :

```
int mat[45][75];
int y,x;

   for (y=0; y<45; y++)    // pour chaque ligne, examiner
       for (x=0; x<75; x++)// chaque élément de la ligne en cours
           mat[y][x]=rand()%2;
```

Le principe est le même pour un nombre quelconque de dimensions. par exemple avec un cube de 10*10*10 :

```
int cube[10][10][10];
int y,x,z;

   for (z=0; z<10; z++)          // pour chaque plan 2D
       for (y=0; y<10; y++)      // pour chaque ligne du plan 2D
           for (x=0; x<10; x++)  // chaque élément de la ligne du
                                 // plan 2D
               mat[z][y][x]=rand()%20;
```

5. Expérimentation tableaux statiques

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/
*****************************************************************
Qu'est ce qu'un tableau ?
- Un tableau est un ensemble d'objets de même type :
  un tableau de int, de float, de double etc.

Qu'est qu'un tableau statique ?
- le fait qu'il soit static signifie qu'il n'est pas dynamique
c'est à dire que sa taille est fixe et que son espace mémoire est
alloué par la machine à la déclaration.

Comment avoir un tableau statique dans un programme ?
Pour définir et déclarer un tableau il faut donner :
- le type des éléments
- un nom pour le tableau
- le nombre des éléments avec l'opérateur crochet (entre crochets)
  soit le formalisme : <type> <nom> < [ constante entière ] >

le nombre des éléments doit toujours être une valeur entière
constante (pas une variable)
```

```
La déclaration est la définition close par un point virgule
Par exemple :
int tab[50];   // déclaration d'un tableau de 50 int
float f[90];   // déclaration d'un tableau de 90 float

Comment accéder à chaque élément du tableau ?

Tous les éléments d'un tableau sont numérotés du premier 0 au
dernier taille tableau-1.  Ce numéro est l'indice de l'élément. Il
indique la place de l'élément dans le tableau (Attention le
premier élément du tableau est à l'indice 0).

Ensuite pour accéder à un élément du tableau il faut utiliser
l'opérateur crochet [ ], par exemple :
int toto[5]; // les indices sont 0, 1, 2, 3, 4. (la taille 5 n'est
             // pas un indice du tableau)

    toto[2] = 123;   // mise à 123 du 3eme int du tableau
    toto[0] = 10;    // mise à 10 du premier int du tableau
    toto[4] = 678;   // mise à 678 du cinquième int du tableau

************************************************************/

#define NBMAX  50

int main()
{

int toto[5];
   toto[2] = 123;
   toto[0] = 10;
   toto[4] = 678;

   printf("toto[%d]=%d\n",2, toto[2]);
   printf("toto[%d]=%d\n",0, toto[0]);
   printf("toto[%d]=%d\n",4, toto[4]);

/*
   //-------------------------------------------------------------
   // possibilité d'initialiser un tableau à la déclaration.
   // Ce qui n'est pas spécifié vers la fin est mis à 0
int toto[10]={1,2,3};
   printf("toto[%d]=%d\n",0, toto[0]);
   printf("toto[%d]=%d\n",1, toto[1]);
   printf("toto[%d]=%d\n",2, toto[2]);
   printf("toto[%d]=%d\n",3, toto[3]);
   printf("toto[%d]=%d\n",4, toto[5]);
*/
   //-------------------------------------------------------------
   // Comment initialiser un tableau de plusieurs millers
   // d'éléments ?
/*
int tab[5000];

   // la boucle for permet facilement de faire défiler tous les
   // indices d'un tableau et ainsi d'accéder à chaque élément
   // quleque soit le nombre :
int i;
   for (i=0; i<5000; i++){
      tab[i]=rand()%256;
```

```
      printf("tab[%4d]=%4d\n",i,tab[i]);
   }
*/
   //-------------------------------------------------------------
   // Utiliser une macro constante pour la taille.
   // Pour la taille du tableau il est préférable d'utiliser une
   // macro constante globale de la forme :
   // #define NBMAX  50
   // au dessus du main() ou dans une librairie personnelle
/*
int tab[NBMAX];
int i;

   // initialisation générateur aléatoire
   srand(time(NULL));

   for (i=0; i<NBMAX; i++){
      tab[i]=rand()%256;
      printf("tab[%4d]=%4d\n",i,tab[i]);
   }

   //-------------------------------------------------------------
   // Trier un tableau. Il existe de nombreux algorithmes qui
   // permettent de trier un tableau, par exemple pour ranger ses
   // valeurs en ordre croissant :
int j,tmp;
   for(i=0; i<NBMAX; i++)
      for (j=i+1; j<NBMAX; j++)
         if (tab[j]<tab[i]){
            tmp=tab[i];
            tab[i]=tab[j];
            tab[j]=tmp;
         }
   for(i=0; i<NBMAX;i++)
      printf("%4d\n",tab[i]);
*/
   //-------------------------------------------------------------
   // Tableaux à plusieurs dimensions.
   // il est posible d'avoir un tableau à plusieurs dimension il
   // suffit de spécifier pour chaque dimmension sa taille, par
   // exemple :
/*
float f[3][4];  // tableau de float à 2 dimensions (une matrice)
                 // revient à avoir 3 tableaux de 4 float.

   // l'accès aux éléments se fait toujours avec l'opérateur
   // crochet [ ], un crochet par dimension :
   f[0][0]=0;
   f[0][1]=1;
   f[0][2]=2;
   f[0][3]=3;
   f[1][0]=4;
   //(...)
   f[2][3]=11;

   // l'utilisation de boucle requiert une variable et une boucle
   // par dimension :
int y, x,cmpt;

   // initialisation
```

```
   for (cmpt=0,y=0; y<3; y++)
      for (x=0; x<4; x++,cmpt++)
         f[y][x]=cmpt;

   // affichage
   for (cmpt=0,y=0; y<3; y++)
      for (x=0; x<4; x++,cmpt++)
         printf("f[%d][%d]=%0.1f\n",y,x,f[y][x]);
*/
/*
   //-------------------------------------------------------------
   // possibilité également d'initialiser un tableau à plusieurs
   // dimensions à la déclaration, 2 solutions :
int tab1[4][2]={ 1,2,3,4,5,6,7,8};
int i,j;
   printf("tab1 :\n");
   for (i=0; i<4; i++){
      for (j=0; j<2; j++)
         printf("%2d",tab1[i][j]);
      putchar('\n');
   }

int tab2[4][2]={ {1,2},
                 {3,4},
                 {5,6},
                 {7,8} };
   printf("tab2 :\n");
   for (i=0; i<4; i++){
      for (j=0; j<2; j++)
         printf("%2d",tab2[i][j]);
      putchar('\n');
   }
   //Remarque : tester en passant la taille dim 2 à 3
   //(ne pas oublier de modifier le test de la boucle 2)
   //et voir résultat
*/
   return 0;
}
```

6. Mise en pratique : base tableaux statiques (non dynamiques)

a. Déclaration de tableaux, accès aux éléments

Exercice 1
Soit un tableau de 5 entiers, Appliquer les instructions suivantes :

1) initialiser le tableau avec des valeurs aléatoires comprises entre 0 et 20.

2) afficher le tableau de façon claire et lisible.

3) Si la première valeur est supérieure à la seconde, permuter les deux valeurs,
si la seconde est supérieure à la troisième permuter,
si la troisième est supérieure à la quatrième permuter,
si la quatrième est supérieure à la cinquième permuter,

A quelle position se trouve la plus grande valeur ?

4) Afficher le tableau pour constater les changements opérés.

Exercice 2
Soit un tableau de 3 entiers et un autre tableau de 5 floats.

1) Initialiser ces tableaux avec des valeurs aléatoires

2) Pour chaque indice comparer les valeurs des deux tableaux et stockez la différence exacte dans un troisième tableau.

3) vérifiez vos résultats par l'affichage.

b. Initialisation de tableaux à la déclaration

Exercice 3
Faire un programme qui déclare un tableau de 10 entiers initialisé avec les valeurs d'indice. Affichez trois positions choisies aléatoirement. A ces positions mettez une valeur comprise entre 0 et 256, affichez la nouvelle valeur à cette position

Exercice 3
Sans préciser le nombre de lettres qu'il contient déclarer un tableau de char initialisé avec un message simple. Choisissez trois positions aléatoires et modifiez les lettres de ces positions.
Mettez le signe '\0' (antislash zéro) à la dernière position de votre tableau et affichez le message avec la fonction printf() et le format %s

Exercice 4
Vous vous lancez dans la programmation d'un tétris (bientôt). Coder les formes T, S et Z, O-carré, I, J et L sachant que pour chacune il y a quatre positions possibles pour les rotations gauche ou droite et que la taille de chaque forme sera de 4 sur 4.

Exercice 5
Faire un programme qui déclare un tableau de 543*896*531 éléments du type que vous voulez.

Tester. Quel résultat obtenez-vous ?

Modifier la taille du tableau de façon à ce que ça marche. Quelle taille fonctionne ?

Le tableau est initialisé à la déclaration (deux ou trois valeurs uniquement). Affichez trois positions choisis aléatoirement.

c. Tableaux à plusieurs dimensions

Exercice 6
Faire un programme qui déclare un tableau à 3 dimensions de 2*2*2 et :
- □ initialiser ce tableau en stockant à chaque position le numéro de la position (1, 2, 3 ...)
- □ afficher le résultat.

Exercice 7
Faire un programme qui déclare deux matrices d'entiers de 2*3 et :
- □ initialisez ces matrices avec des valeurs aléatoires entre 0 et 20

- pour chaque position comparez les valeurs stockées et stockez la plus grande dans une troisième matrice.
- affichez les trois matrices (sur 6 lignes uniquement)

Exercices 8
Faire un programme qui déclare une matrice de 100*450 floats.
- initialisez 3 positions différentes choisies aléatoirement dans la matrice avec des valeurs entre 0 et 2
- affichez les trois positions et les valeurs aux trois positions.

d. Boucles et tableaux

Exercice 9
Soit un tableau de 2000 float
- l'initialiser avec des valeurs aléatoires comprises entre 0 et 10
- afficher le tableau avec uniquement 2 chiffres après la virgule

Exercice 10
Soit un tableau de 150 int,
- initialisez aléatoirement le tableau avec uniquement des 0 ou des 1.
- affichez le tableau en 10 colonnes de 3 caractères et 15 lignes. La taille des colonnes est spécifiée entre le % et le d du format de la façon suivante "%3d".

Exercice 11
Soit une matrice de 15 lignes par 10 colonnes :
- initialisez le tableau avec des valeurs croissantes aléatoires.
- affichez le résultat. La taille des colonnes est spécifiée comme dans l'exercice précédent avec un intervalle de un caractère entre chaque nombre.

Exercice 12
Soit une matrice de 15*15. Faire un damier et afficher le résultat.

Exercice 13
Remplir un tableau avec des valeurs aléatoires et opérer une rotation des valeurs afin que ce qui est à la position 0 passe à la position 1, ce qui est à la position 1 passe à la position 2 et ainsi de suite jusqu'au dernier qui vient à la position 0.
Faire le même exercice avec une matrice : la valeur de chaque position passe à la position suivante jusqu'à la dernière qui vient en première position.

Exercice 14
Remplir un tableau avec des valeurs aléatoires, afficher le tableau puis :
- affichez les deux plus grandes valeurs
- affichez la plus petite valeur et sa position dans le tableau
- affichez les valeurs du tableau en ordre croissant sans modifier le tableau
- modifiez le tableau afin que toutes ses valeurs soient triées en ordre décroissant et affichez le résultat.

Exercice 15
Soit un groupe d'enfants dans une cours d'école. Ils veulent jouer à chat et pour désigner le chat ils utilisent la méthode "am stram gram pic et pic et co le gram bou ret bou ret ra ta tam am stram gram". En répétant de façon rythmée cette phrase l'un d'entre eux désigne, à chaque syllabe une personne du groupe, toujours selon le même ordre et en tournant toujours dans le même sens, celle désignée avec la

dernière syllabe ne sera pas le chat et est sortie du groupe. L'opération est recommencée jusqu'à ce qu'il n'en reste plus qu'une qui est le chat.
Faire un programme où le groupe des enfants est représenté par un tableau et utiliser cette méthode pour désigner le chat qui correspondra au dernier indice restant du tableau.

Exercice 16
Un poète maudit vit dans un grenier. Il manque deux tuiles au toit et il pleut. Le poète a placé une bassine sous chaque trou. Les bassines ont la même taille et chaque bassine se remplit goutte à goutte en fonction d'un temps aléatoire. Laquelle des deux bassines est remplie en premier et en combien de temps ? Faire une simulation en prenant comme bassine deux tableaux, pour chaque goutte le temps écoulé depuis la dernière goute de la même bassine est stocké.

Exercice 17
Le lave vaisselle est en panne et la vaisselle s'accumule. Le plongeur s'active, il essaie de nettoyer les assiettes et autres objets dés qu'ils arrivent mais ils arrivent très vite et s'empilent à côté de lui. Définissez quatre types d'objets (par exemple assiette plate, creuse, à désert et plat). A partir d'un tableau simuler la situation de la pile de vaisselle. Lorsque l'on appuie sur une touche des objets sont ajoutés et lorsque l'on appuie sur une autre touche du clavier des objets sont retirés de la pile. Lorsque l'on appuie sur enter le contenu de la pile des objets est affiché.

Exercice 18
Nous sommes à un carrefour routier, il y a un feu qui est rouge ou vert. Faire une simulation simplifiée de la constitution d'une file de voiture à l'un des feux. Lorsque le feu est rouge les voitures qui arrivent sont ajoutées à la file. Lorsque le feu passe au vert les voitures en tête de la file partent une à une et libèrent leur place. Mettre en place ce système de file en utilisant un tableau.

E. Exemples d'utilisations de tableaux

1. Chaines de caractères

Une chaine de caractères (string en anglais) est une suite de caractères stockée dans un tableau de char et terminée par le caractère '\0'.

Le '\0' marque la fin de la chaine quelque soit la taille du tableau mais le nombre total de caractères dans la chaine ne peut pas dépasser la taille du tableau.

Toutes les fonctions de traitement de chaines de caractères s'appuie sur le '\0' final pour trouver la fin de la chaine et jamais sur la taille du tableau.

Un tableau de char peut être initialisé à la déclaration avec une chaine de caractères, Soit par exemple la séquence :

```
char s[100]="bonjour\n"; // 9 caractères dans la chaine : '\0'
implicite
                         // le reste du tableau n'est pas utilisé
   printf(s);
   s[3]='s';
   s[5]='i';
```

```
   printf(s);
```

La chaine "bonjour\n" est une constante chaine de caractères (telle quelle elle n'est pas modifiable). Un '\0' est ajouté par la machine à toutes les constantes chaine de caractères. C'est pourquoi cette chaine à 9 caractères au total : 7 lettres, 1 retour chariot et le '\0' final. Le tableau s est initialisé avec cette chaine de 9 caractères mais c'est le seul cas où l'utilisation de l'opérateur d'affectation est utilisable en C pour une copie de chaine de caractères. Sinon pour copier deux chaines de caractères il faut utiliser la fonction strcpy().

En cas d'oubli du '\0' final le résultat est incertain, ça plante où il y a un affichage bizarre avec des signes inattendus (ce qui traine en mémoire avant de tomber sur un '\0'). Vous pouvez tester par exemple :

```
char s1[]={'a','b','c','d'}; // manque le '\0' final : résultat
                             // incertain
   printf("test sans \\0 : %s\n",s1);
```

Le tableau s1 est un tableau de quatre caractères mais ce n'est pas une chaine de caractères parce qu'il manque le '\0' final.

Il existe de nombreuses fonctions de traitement de chaines fournies en standard par la librairie string.h du langage C. Notamment la fonction fgets() qui permet à l'utilisateur d'entrer une chaine de caractères pendant le fonctionnement du programme :

```
int main()
{
char recup[100];
   printf("entrer une phrase :\n");
   fgets(recup,100,stdin);
   printf("phrase entree : %s\n",recup);
   return 0;
}
```

La fonction fgets() prend en paramétre 1 le tableau dans lequel est récupérer la chaine, en paramètre 2 la taille maximum du tableau, en paramètre 3 le fichier dans lequel l'entrée a lieu (stdin est un fichier créé et ouvert automatiquement en mode console pour toutes les entrées utilisateur)

Pour copier deux chaines de caractères il faut utiliser la fonction strcpy() :

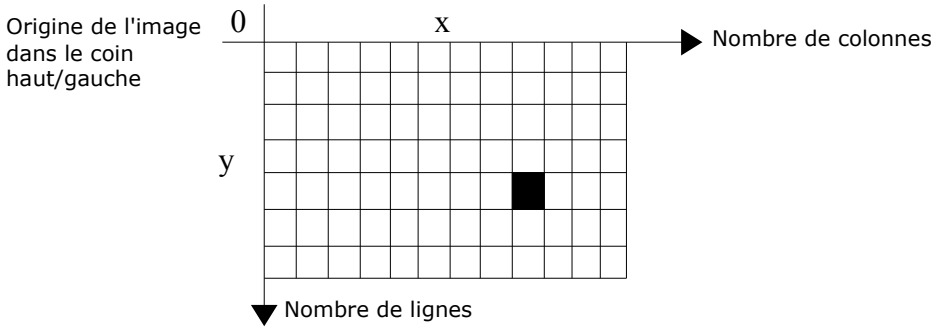
```
int main()
{
char recup[100], copie[100];

   printf("entrer une phrase :\n");
   fgets(recup,100,stdin);
   strcpy(copie,recup);
   printf("copie : %s\n",copie);
   return 0;
}
```

Le premier caractère de strcpy() est la chaine pour la copie et le second la chaine à copier.

2. Image bitmap

Une image numérique est un tableau à deux dimensions de nombres. Chaque position de la matrice est appelée un pixel de l'image et sa valeur correspond au codage d'une couleur.

Le nombre total de couleurs accessibles pour l'image ainsi que la taille de l'image en mémoire vont dépendre du nombre de bits utilisés pour le codage de la couleur (8, 15, 16, 24, 32 bits).



En noir un pixel à une position de la matrice image et auquel correspond une couleur codée par un entier. En général 0 pour désigner le noir.

3. Stocker des données localisées

Le voyageur dans un train

Imaginons de faire la liste des noms des voyageurs assis dans les trains de plusieurs gares sans perdre les informations qui permettent de localiser chacun des voyageurs. Admettons que nous ayons 4 gares, un max de 100 trains par gare, un max de 25 wagons par train, un max de 60 places par wagon. Une structure de donnée peut être :

```
char stock_noms[4][100][25][60][100];
```

Je prends mon train dans la gare 3, le train n°33, le wagon n°17, la place n°42. L'affectation n'est pas possible avec les chaines de caractères et le nom est stocké par copie avec la fonction strcpy() :

```
strcpy(stock_noms[2][33][17][42], "Frédéric Drouillon");
```

Pour chaque gare on accède à chaque train, pour chaque train on accède à chaque wagon et pour chaque wagon à chaque place.

Remarque :

Dans cet exemple nous pourrions utiliser des macro constantes pour les noms de gare (ici les quatre grandes gares parisiennes : du nord, de l'est, de Lyon, et Montparnasse):

```
#define NORD                0
#define EST                 1
#define LYON                2
#define MONTPARNASSE        3
```

Ce qui permet d'écrire ensuite :

```
strcpy(stock_noms[LYON][33][17][42],"Frédéric Drouillon");
```

ou encore un enum ce qui permet aussi de tout regrouper :

```
enum voyageurs{
   NORD,
   EST,
   LYON,
   MONTPARNASSE,
   NB_GARE      ,
   NB_TRAIN=100,
   NB_WAGON=25,
   NB_PLACE=60,
   TAILLE_NOM=100
};

char stock_noms[NB_GARE][NB_TRAIN][NB_WAGON][NB_PLACE]
[TAILLE_NOM];
```

4. Expérimentation : Exemple d'utilisation de chaines de caractères

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

/
*****************************************************************
UTILISATION CLASSIQUE DE TABLEAUX : chaines de caractères

Qu'est ce qu'une chaine de caractères ?
une suite de caractères stockée dans un tableau de char et
terminée par le caractère '\0'
Le '\0' marque la fin de la chaine et est indispensable.Toutes les
fonctions de traitement des chaines de caractères s'appuient sur
cette marque de fin pour leurs traitements. S'il est ommis il
risque d'y avoir débordement de tableau et le programme va planter
ou avoir un comportement incertain. Sans le '\0' final, il ne
s'agit pas d'une chaine mais uniquement d'un tableau de
caractères.

*****************************************************************/
int main()
{
char s[100]="bonjour\n"; // 9 caractères dans la chaine : '\0'
implicite
                        // le reste du tableau n'est pas utilisé
   printf(s);
   s[3]='s';
   s[5]='i';
   printf(s);
/*
   //-------------------------------------------------------------
   // si pas de \0 résultat incertain, plantage ou aléatoire
char s1[]={'a','b','c','d'};
   printf("test sans \\0 : %s\n",s1);
*/
/*
   //-------------------------------------------------------------
   // la taille de la chaine définie par l'initalisation à la
```

```
    // déclaration
char s2[]="tata fait du velo"; // 17 caractères + \0

   // la fonction strlen renvoie le nombre de caractère d'une
   // chaine sans compter le \0 final :
   printf("s2 a %d caracteres\n",strlen(s2));
*/
/*
   //-------------------------------------------------------------
   // pour entrer une chaine de caractères :
   // fonction fgets(le tab pour recup, la taille max, le fichier
   // source)
char recup[100];
   printf("entrer une phrase :\n");
   fgets(recup,100,stdin);
   printf("phrase entree : %s\n",recup);
*/
/*
   //-------------------------------------------------------------
   // pour copier une chaine de caractères :
   // fonction strcpy(tab pour copie, chaine à copier)
char copie[100];
   strcpy(copie, recup);
   printf("voici la copie : %s\n",copie);
*/
   return 0;
}
```

5. Mise en pratique tableaux

a. Chaines de caractères

Exercice 1
A partir d'un menu proposé à l'utilisateur faire le programme suivant :

- Saisir une chaîne de caractères
- Compter la longueur d'une chaîne
- Convertir la chaîne saisie en majuscules. Seuls les caractères en minuscule sont modifiés les autres sont évités.
- Convertir la chaîne saisie en minuscules
- Comparer 2 chaînes à saisir. Il s'agit de dire si elles sont identiques et laquelle est la première dans l'ordre lexicographique.
- Concaténer 2 chaînes dans la première (attention aux débordements)
- Crypter une chaîne en appliquant un décalage circulaire dont la valeur est entrée par l'utilisateur
- Décrypter une chaîne de caractères cryptée en appliquant un décalage circulaire inverse à partir d'une valeur entrée par l'utilisateur.

Exercice 2
La commande suivante arrive par courriel :

"Bonjour,
Je souhaiterais un générateur de mots de passe aléatoires avec les possibilités suivantes :

- choisir le nombre de mots de passe à générer

- □ choisir la longueur du mot de passe (de 6 à 30 caractères max)
- □ choisir les types de caractères utilisés (minuscules, majuscules, chiffres, signes)
- □ vérification de l'usage de tous les types de caractères sélectionnés
- □ Exclure les caractères O et I qui prêtent à confusion

...Merci, ...Cordialement,..."

Faire une proposition.

Exercice 3

M. A est une lettre qui se déplace sur l'ensemble d'une aire de jeu définie. Autour de lui sont placées au hasard des lettres de couleurs que M.A peut ramasser. En fait une phrase (ou un mot) est disséminée et il doit la reconstituer. Chaque bonne lettre trouvée vient s'adosser derrière lui de sorte que la phrase constitue progressivement une chenille. Il s'agit de reconstituer la phrase entière. Le programme pourra tenir en réserve plusieurs phrases afin de pouvoir faire plusieurs parties. Le choix des phrases est aléatoire. Programmer.

Exercice 4

Le jeu du pendu. Le programme affiche un mot de sept lettres et le joueur doit trouver ce mot. Pour ce faire il propose un mot. Si des lettres coïncident entre le mot a trouver et le mot proposé elles sont affichées à leur bonne place (les lettres non trouvées sont remplacées par des tirets). L'objectif est de trouver le mot avec un minimum de proposition.

Exercice 5

Faire un générateur automatique de texte. Le principe est de créer des modèles de phrases. C'est-à-dire des "moules à phrases" que l'on désigne par le terme de "protophrases". Une protophrase est construite à partir d'un enchaînement de catégories de mots ou d'expressions. La protophrase de base est simple, du type : Nom / Adjectif / Verbe / Nom. Chaque catégorie est un ensemble de mots du type correspondant. Pour faire une phrase il suffit ensuite, selon le modèle initial, de faire un tirage aléatoire dans chaque catégorie et d'assembler le résultat.

Pour chaque catégorie il y a une librairie de mots ou d'expressions rangés dans un tableau directement dans le code source du programme. Bien sur des mots fixes peuvent être ajoutés dans le protophrase afin d'augmenter les possibilités expressives, par exemple : Nom / Verbe / AVEC / Nom / SUR / Nom. Les mots "avec" et "sur" sont fixes dans la protophrase.

Exercice 6

Remplir une matrice de lettres et, à partir d'une liste de mots que vous choisissez, regarder si des mots sont présents par hasard. Le programme indique les mots présents. Attention, les lettres du mot peuvent être consécutives mais il est plus probable qu'elles soient dispersées. En revanche vous considérerez qu'elles doivent arriver dans l'ordre. Par exemple le mot TATA est dans la ligne ci-dessous :
resdarTferdorAafdcvmTnfertuhgfdsqArfdyui
Pour l'affichage utilisez les fonctions fournies en annexe.

Exercice 7

Dans un programme faire deux listes de mots. Choisir aléatoirement des mots dans ces deux listes sélectionnée aléatoirement également et les recopier les uns à la suite des autres dans une matrice de lettres. Parcourir ensuite la matrice et pour chaque mot retrouvé dire s'il fait parti de la liste 1 ou de la liste 2 et reconstituez l'alternance du tirage entre les deux listes.

b. Image, terrain de jeux

Exercice 8
Dans un nouveau programme, déclarer une matrice de 10 lignes par 20 colonnes.
Nous allons considérer cette matrice comme une image en 256 couleurs, c'est à dire pour laquelle il y a un maximum de 256 couleurs.

1) Remplir l'image avec des valeurs aléatoires comprises entre 0 et 255 (bornes comprises)

2) Afficher cette image à l'écran en respectant son aspect rectangulaire. Vous pouvez utiliser la fonction gotoxy() et les fonctions pour la couleur fournies en annexe.

3) Calculer de l'histogramme de l'image.
L'histogramme d'une image étudie la répartition statistique de chaque valeur de couleur dans une image en 256 couleurs. Son principe consiste, dans un tableau histogramme de 256 cases de type entier, à compter combien l'image contient de pixels de la couleur 0 et à stocker cette valeur dans la case 0 du tableau histogramme, puis combien l'image contient de pixels de couleur 1 à ranger le nombre obtenu dans la case 1 du tableau histogramme ... ainsi de suite pour toutes les couleurs jusqu'à la couleur 255 comprise.
Une fois l'histogramme calculé, afficher le résultat de manière lisible à l'écran.

4) Binarisation de l'image
La binarisation d'une image consiste, dans une image secondaire afin de ne pas modifier l'image originale, à mettre à 0 tous les pixels de l'image originale inférieurs à une valeur seuil entrée par l'utilisateur, et à mettre à 255 tous les pixels de l'image originale supérieurs ou égaux à cette valeur seuil. Une fois cette opération effectuée afficher le résultat.

Exercice 9
Dans un jeu vidéo, la technique du "tile map" sert à construire des terrains de jeu, des paysages ou différents levels. Le principe est simple. A la base on a des petites images élémentaires : par exemple gazon, mur, eau, feu. Chaque petite image correspond à un numéro, par exemple 0 pour gazon, 1 pour mur, 2 pour eau, 3 pour feu. Ensuite dans une matrice est réalisé un dessin à partir de ces nombres. Dans le jeu lorsque le terrain est appelé, l'image est construite à partir de la matrice et affichée à l'écran.

- □ Initialisez une matrice de façon à faire un terrain de jeu de 10 par 15.
- □ Affichez votre terrain avec la fonction gotoxy() et les fonctions pour la couleur fournies en annexe
- □ placez des objets de façon aléatoire sur votre terrain. Les objets ne peuvent pas être sur les murs, ils ne peuvent être posés que sur le gazon. Attention, les

objets ne sont pas stockés dans la matrice du terrain de jeu. Ensuite affichez ces objets

- □ après un temps qui lui est propre chaque objet change de place. Dans une boucle d'évènements, faites changer de place les objets chacun à son rythme.

Exercice 10
Bientôt Noël peut-être ? Fabrication de guirlandes électriques clignotantes pour sapin. Faire un programme qui simule une guirlande électrique avec plusieurs lampes de couleurs différentes.

Exercice 11
Une quinzaine de petits carrés (obtenus avec l'affichage d'un espace pour lequel la couleur de fond est spécifiée) se déplacent à des vitesses différentes dans un espace strictement délimité. Lorsqu'ils arrivent sur un bord ils rebondissent et partent dans une autre direction.

Exercice 12
Dans un terrain construit à partir d'une matrice déplacer un player. Le player respecte les murs. S'il y a des objets sur le terrain le player les ramasse et marque des points.

Exercice 13
Dans un zone de jeu, une chenille avance pilotée par les touches flèche. Elle s'allonge de un tous les 5 tours. Faire le programme. Si ce n'est pas fait, compléter le programme de façon à ce qu'elle ne puisse pas se marcher dessus.

Exercice 14
En utilisant la fonction gotoxy() faire un programme de neige qui tombe de haut en bas de l'écran. Trouver une solution pour l'arrivée des flocons : est ce qu'ils fondent ? est ce qu'ils s'accumulent ? Si oui jusqu'où ? …

c. Localisation de données via plusieurs dimensions

Exercice 15
Mesures dans un laboratoire. Afin d'étudier des effets d'une molécule un laboratoire organise une expérience avec 150 personnes. Ces personnes sont réparties en trois groupes de 50, l'expérience dure 3 mois avec une série de mesures spécifiques pour chaque mois et pour chaque groupe :

- □ Le groupe 1 sert de référence et ne reçoit pas de traitement.
- □ Le groupe 2 reçoit un « placebo » sans principe actif biologiquement.
- □ Le groupe 3 reçoit le principe actif

Il y a quatre mesures différentes toutes les 60 secondes, 24h sur 24h et la date doit être conservée (mois, jours). Les mesures sont température, taux de lymphocytes (globules blancs), taux d'hématies (globules rouges) et temps exacte où la mesure est effectuée. Comment stocker les résultats ? Faire le programme qui permet d'entrer chacune des mesures.

F. Tableaux et structures

1. Tableau comme champ dans une structure

Une structure peut contenir des tableaux. L'accès au tableau se fait comme pour n'importe quel autre champ avec l'opérateur point. Soit par exemple la structure test suivante :

```
typedef struct{
   char nom[80];
   float calc;
   int stock[10];
}test;
```

Dans le programme suivant une struct test est initialisée et affichée:

```
int main()
{
test t1;
int i;

   strcpy(t1.nom,"Michael");
   printf("Nom : %s\n",t1.nom);
   t1.calc=(float)rand() / RAND_MAX;
   printf("Calcul : %f\n",t1.calc);

   for (i=0; i<10; i++){
       t1.stock[i]=rand()%256;
       printf("Valeurs %d stockees : %d\n", i, t1.stock[i];
   }
   return 0;
}
```

Au départ, déclaration de la structure t1, ensuite :

- initialisation et affichage du champ nom
- initialisation du champ calc avec une valeur flottante aléatoire entre 0 et 1 et affichage de la valeur.
- initialisation du champ stock, un tableau de 10 entiers avec des valeurs aléatoires et affichage du tableau. Une boucle est utilisée pour accéder aux éléments du tableau.

2. Tableau de structures

On peut avoir un tableau dans une structure et il est possible également d'avoir un tableau de structures. Soit en global la définition et déclaration de la structure suivante :

```
struct pix{
   int x, y, color;
};
```

Dans un programme nous déclarons une struct pix et un tableau de trois struct pix. Elles sont ensuite initialisées et affichées, d'abord la structure seul puis le tableau de structure :

```
int main()
{
struct pix  p, tab[3];
int i;
   // initialisation structure seule
```

```
   p.x=0;
   p.y=50;
   p.color=255;
   // affichage structure seule
   printf("x=%d,y=%d,color=%d",p.x,p.y,p.color);

   //initialisation du tableau de structures
   for (i=0; i<3; i++){
       tab[i].x=0;
       tab[i].y=50;
       tab[i].color=255;
       //affichage de chaque structure
       printf("tab[%d].x=%d,y=%d,color=%d",
              i,tab[i].x,tab[i].y,tab[i].color);

   }

   (. . .)

}
```

Une boucle for est utilisée afin d'accéder à chaque structure du tableau tab. Dans cet exemple les trois structures dans le tableau sont initialisées avec les mêmes valeurs, les champs x prennent 0, y 50 et color 255.

Autre exemple, dans un jeu vidéo la structure suivante caractérise des ennemis :

```
typedef struct{
   char nom[80];     // nom et type de l'ennemi
   float nuisance;   // potentielle de nuisance
   float x,y,px,py;  // position et déplacement
   int force[10];    // 10 possibilités de force pour les combats
}t_ennemi;
```

Nous avons avec une fonction d'initialisation du genre :

```
t_ennemi init_ennemi(void)
{
char* lesNoms[4]={"Bretelgunch","Athanase","Belzebon",
                  "Trolympique"};
t_ennemi e;
int i;
   // un nom au hasard parmi les quatre
   strcpy(e.nom, lesNoms[rand()%4]);
   e.nuisance= rand()/(float)RAND_MAX;
   e.x = (rand()/(float)RAND_MAX) * 1024;    // x de 0 à 1024
   e.y = (rand()/(float)RAND_MAX) * 768;     // y de 0 à 768
   e.px = ((rand()/(float)RAND_MAX) * 10)-5; // de -5 à +5
   e.py = ((rand()/(float)RAND_MAX) * 10)-5; // de -5 à +5
   for (i=0; i<10; i++)
       //les 10 indicateurs de force entre 0 et 100
       e.force[i]=rand()%100;
   return e;
}
```

et aussi une fonction d'affichage :

```
void affiche_ennemi (t_ennemi e, int number)
{
int i;
   printf("Ennemi %d :\n", number);
```

```
   printf("\t- nom                    : %s \n", e.nom);
   printf("\t- nuisance               : %.2f \n", e.nuisance);
   printf("\t- pos horizontale        : %.2f \n", e.x);
   printf("\t- pos verticale          : %.2f \n", e.y);
   printf("\t- deplacement horizontale : %.2f \n", e.px);
   printf("\t- deplacement verticale   : %.2f \n", e.py);
   for(i=0; i<10; i++)
       printf(force[%d]=%d\n",e.force[i]);
}
```

Le programme suivant déclare un tableau struct ennemi et l'initialise :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NB_ENNEMI     100

// définition de la structure ennemi ici.

// déclaration des fonctions ici

int main()
{
t_ennemi E[NB_ENNEMI];
int i;
   srand(time(NULL));
   for (i=0; i<NB_ENNEMI; i++){
       E[i]=init_ennemi();
       affiche_ennemi(E[i],i);
   }
   return 0;
}

// définir les fonctions ici
```

3. Différences entre tableaux et structures

Il y a trois différences entre un tableau et une structure :

1) Le tableau est un ensemble indicé d'objets de même type, on accède aux éléments avec l'opérateur [ ]. La structure est un ensemble d'objet qui peuvent être de types différents et il n'y a pas d'indicage, on accède aux éléments avec l'opérateur **.** point.

2) le tableau n'est pas une "left value", c'est à dire qu'il n'est pas possible d'affecter quelque chose à un tableau. La structure est une left value et il est possible d'affecter à une structure une structure du même type (copie).

3) Le tableau en paramètre de fonction est automatiquement converti en pointeur (ce point est abordé dans la partie tableaux et fonctions) et lors de l'appel de la fonction il reçoit l'adresse mémoire du tableau passé en argument. C'est un passage d'argument par référence à l'adresse mémoire. La structure en paramètre de fonction est simplement une variable locale à la fonction à laquelle est affectée

une valeur au moment de l'appel, c'est un passage par valeur sans référence à une adresse mémoire.

4. Mise en pratique : tableaux de structures

Exercice 1
Soit les deux structures :

```
struct date{
          int jour, mois, annee;
};
struct personne{
          char nom[80];
          struct date date_embauche;
          struct date date_poste;
};
```

- Ecrire une fonction pour initialiser une struct personne.
- Ecrire une fonction d'affichage afin d'obtenir :

```
nom : TOTO
date embauche (jj mm aa) : 09 09 10
date poste = date embauche ? (O/N)  : O

OU

nom : TOTO
date embauche (jj mm aa) : 09 09 10
date poste = date embauche ? (O/N)  : N
date poste (jj mm aa) :  01 01 11
```

Exercice 2
Soit dans un jeu vidéo une entité en mode console. Elle est définie par une position, un déplacement, un type (rampant, grouillant, serpentant, plombant, assommant), une couleur et une lettre. L'entité a également un nom et une série de taux : taux d'agressivité, de colère, de convoitise, de faim, de peur.
Définir la structure de données pour coder une entité. Comment avoir 100 entités ? Faire une fonction d'initialisation, une fonction de mise à 0 (reset) et une fonction d'affichage. Tester dans un programme avec un menu : quitter, afficher, initialiser, reset. Ajouter une fonction pour animer les entités (utiliser la fonction void gotoxy(int x, int y) fournie en annexe) et quelques fonctions qui permettent aux entités d'établir des relations entre elles.

Exercice 3
Faire carnet de NBMAX rendez-vous. Un rendez-vous sera défini par :

- Un libellé
- Un lieu de rendez-vous
- Une date (choisissez un format qui permettra le tri)
- Un horaire de début
- Un horaire de fin
- Une catégorie (bureau, personnel, loisirs etc.)
- Tout autre champ pouvant vous sembler utile

1) définir le ou les types nécessaires à une bonne structure de données
2) Dans un menu donnez les possibilités de :
   - saisir un rendez-vous

- □ afficher tous les rendez-vous
- □ afficher une catégorie de rendez-vous
- □ supprimer un rendez-vous
- □ supprimer automatiquement des rendez-vous en doublon
- □ ranger en ordre chronologique les rendez-vous
- □ quitter

Exercice 4
Soit la structure suivante :

```
struct point{
    int x, y, color;
};
```

Dans un programme déclarer un tableau de NB_POINT (par exemple 10).

- □ Faire une fonction d'initialisation. Cette fonction initialise chaque point soit au hasard soit par des valeurs entrées par l'utilisateur.
- □ Faire une fonction d'affichage
- □ Faire une fonction qui permet de modifier les valeurs d'un des points du tableau.
- □ Faire une fonction de mise à zéro de toutes les valeurs du tableau.
- □ Donner un menu utilisateur avec les commandes : quitter, initialiser, afficher, modifier, mise à 0

Exercice 5
Faire un programme qui permet d'afficher le signe du zodiaque à partir d'un jour et d'un mois de naissance. Le jour est entré sous forme de nombre et le mois sous forme de chaine de caractères. Voici les différents signes avec leur période :

Capricorne	23 décembre - 19 janvier
Verseau	20 janvier - 19 février
Poisson	20 février - 20 mars
Bélier	21 mars - 19 avril
Taureau	20 avril - 20 mai
Gémeau	21 mai - 20 juin
Cancer	21 juin - 21 juillet
Lion	22 juillet - 22 aout
Vierge	23 aout - 22 septembre
Balance	23 septembre - 22 octobre
Scorpion	23 octobre - 21 novembre
Sagittaire	22 novembre - 22 décembre

Exemple d'exécution :

```
donner votre jour et votre mois de naissance :
11 july
*** erreur de nom de mois****
donner votre jour et votre mois de naissance :
16 janvier
vous êtes né sous le signe : capricorne
```

Exercice 6
Écrire un programme qui traduit une phrase entée au clavier en en morse. Les caractères susceptibles d'être codés sont les 26 lettres de l'alphabet, les chiffres de 0 à 9 et le point.
Si le texte contient d'autres caractères que ceux-ci le programme affichera des points d'interrogation à la place du code morse. Voici les codes morses :

A	.-	B	-...	C	-.-.	D	-..	E	.
F	..-.	G	--.	H	....	I	..	J	.---
K	-.-	L	.-..	M	--	N	-.	O	---
P	.--.	Q	--.-	R	.-.	S	...	T	-
U	..-	V	...-	W	.--	X	-..-	Y	-.--
Z	--..	0	-----	1	.----	2	..---	3	...--
4	....-	5	.....	6	-....	7	--...	8	---..
9	----.	. point	.-.-.-						

Chaque code est terminé par un ou deux espaces, un espace par des points d'interrogation, par exemple "le langage C" donne :

.-.. . ????? .-.. .- -. --. .- --. . ????? -.-.

Le programme quitte uniquement si l'utilisateur le souhaite.

Exercice 7
A partir des données de l'exercice précédent, faire un traducteur dans l'autre sens du morse vers le langage normal. Le programme donne un menu : entrer un message en morse, générer un pseudo message en morse (génération aléatoire du message), quitter.
Chaque message est traduit et affiché.

Exercice 8
Réaliser un programme établissant une facture pour une commande de plusieurs articles. Pour chaque article de la commande, l'utilisateur fournit la quantité et un numéro de code à partir duquel le programme retrouve à la fois le libellé et le prix unitaire. Le programme refuse les codes inexistants. A la fin il affiche un récapitulatif tenant lieu de facture.

Les informations relatives aux différents articles sont définies en globale dans le source du programme. Utilisez un catalogue de supermarché pour faire la liste et inventez les codes.

1) Le programme propose un menu avec :
   - □ nouvelle commande
   - □ afficher la liste des articles disponibles

Prévoir une fonction de recherche des informations relatives à un article à partir de son numéro de code et une fonction d'affichage de la facture récapitulative.

Exemple d'exécution :

```
ARTICLE                 NBRE          P-UNIT        MONTANT

centrifugeuse           33            47.29         1560.57
Grille-pain             12            35.84         430.08
Four Raclette 6p        6             51.33         307.98

TOTAL                                               2298.63
```

2) Au moment de la commande il y a vérification que le nombre d'articles demandés est disponible sinon l'utilisateur est invité à modifier sa commande. Le stock disponible est mis à jour après chaque commande.

G. Tableaux et fonctions

1. Utiliser un tableau déclaré en global

Comment utiliser des tableaux avec des fonctions renvoie à la visibilité (accessibilité) des variables dans un programme : le tableau pourra être déclaré en global avec des fonctions écrites pour lui seul et sans paramètre mais il pourra aussi être déclaré en local par exemple dans le main() avec des fonctions généralisées qui pourront être utilisées avec différents tableaux.

Pour rappel, toutes les variables peuvent être déclarées en local ou en global :

- les variables sont dites "locales" à la fonction dans laquelle elles sont déclarées. C'est à dire qu'elles sont visibles (accessibles) uniquement dans le bloc de la fonction et dans tous ses sous blocs imbriqués. Dans ce cas les valeurs des variables peuvent circuler grâce aux paramètres d'entrées et au mécanisme de retour (return) des fonctions
- Mais il est possible de déclarer des variables au niveau fichier en dehors de tout bloc, au dessus du main(). Dans ce cas la variable est accessible de tous les blocs et toutes les fonctions dans le fichier, sans avoir à utiliser les paramètres d'entrée ou la valeur de retour. La déclaration en global est utilisée pour les définitions de type notamment les structures, pour des valeurs constantes (#define, enum …), pour les déclarations de fonction et pour quelques variables essentielles afin de simplifier l'écriture de petits programmes (moins de 500 lignes de code). En aucun cas cette propriété n'est utilisée si la "globalité" de la variable n'est pas justifiée. Mal utilisée cette propriété risque de nuire au développement (en général le développement se trouve rapidement paralysé au delà de 1000 lignes de code).

Le programme suivant illustre comment utiliser cette propriété pour un tableau de données dans un programme :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// valeur constante définie en globale
```

```
#define NBMAX  50

// tableau déclaré en global, accessible de partout dans le
programme
int tabGlo[NBMAX];

// déclaration des fonctions définies en dessous du main()
void  init_tabGlobal    (void);
void  affich_tabGlobal  (void);

/*******************************************************
*******************************************************/
int main()
{
   init_tabGlobal();   // appel fonction d'initialisation sans
paramètre
   affich_tabGlobal();// appel fonction d'affichage sans paramètre
   return 0;
}
/*******************************************************
Les deux fonctions peuvent agir sur le tableau tabGlo sans
le passer en paramètre parce qu'il est déclaré en  global.
En revanche elles ne peuvent travailler que sur ce tableau.
*******************************************************/
void init_tabGlobal()
{
int i;
   for (i=0; i<NBMAX; i++)
      tabGlo[i]=rand()%256; // initialisation du tableau avec des
                            // valeurs aléatoires comprises entre
                            // 0 et 255
}
/*******************************************************
*******************************************************/
void affich_tabGlobal()
{
int i;
   for (i=0; i<NBMAX; i++){   // affichage du tableau
      printf("%4d",tabGlo[i]);
      if (i%10==9)            // par ligne de 10
         putchar('\n');
   }
}
```

2. Tableau en paramètre de fonction

a. Précision sur le type tableau

Soit par exemple n'importe où dans un programme la déclaration :

```
int tab[50];
```

Quelle est la valeur numérique de tab ? Qu'est ce que tab pour la machine ?

- tab c'est l'adresse mémoire du bloc alloué pour le tableau, c'est à dire l'adresse du premier élément du tableau et ils sont tous consécutifs en mémoire (pas de trous dans le bloc).
- Passer un tableau à un paramètre de fonction ça suppose passer une adresse mémoire au paramètre.

- Le type de variable qui prend pour valeur des adresses mémoire est le type pointeur.
- Donc, pour passer un tableau à une fonction il lui faudra un pointeur en paramètre

b. La variable pointeur

Le pointeur est simplement une variable qui prend pour valeur des adresses mémoire. D'une façon générale dans un programme une variable pointeur est spécifiée avec l'opérateur étoile de la façon suivante : si T est un type, T* est le type pointeur sur T. Par exemple :

```
char*s;
est un pointeur de caractère, cette variable peut contenir  des
adresses mémoire de variables de type char

int*t;
est un pointeur d'entier, cette variable peut contenir  des adresses
mémoire de variables de type int

float*f;
est un pointeur de float, cette variable peut contenir  des adresses
mémoire de variables de type float
```

A partir de ces pointeurs on pourrait avoir les affectations suivantes :

```
// soit 4 variables déclarées comme suit :
int toto=70;
char tata='A';
float titi=7.5;
int tab[10];

   // nous récupérons l'adresse de chacune des variables avec
   // l'opérateur & et nous affectons cette adresse à un
   // pointeur :
   s=&tata;
   f=&titi;
   t=&toto;

   // dans le cas du tableau, tab est déjà une adresse il n'y a
   // donc pas besoin de l'opérateur & et nous pouvons écrire :

   t=tab;       // t (un int*) prend l'adresse du tableau tab
                // (tableau d'entiers, adresse du premier élément)
```

c. En paramètre de fonction le tableau est converti en pointeur

Comme un pointeur permet de récupérer une adresse mémoire et qu'un tableau est une adresse mémoire les tableaux en paramètre de fonction sont des pointeurs.

Tableau à une dimension

En paramètre de fonction, un pointeur destiné à recevoir un tableau peut prendre trois formes équivalentes qui sont, soit T un type et tab un tableau :

```
T *tab
T tab[ ]
T tab[NB_ELEMENT]
```

Par exemple :

```
void init (int*t)
{
int i;
   for (i=0; i<10; i++){
       t[i]=rand()%256;
       printf("t[%d]=%d\n",i,t[i]);
   }
   putchar('\n');
}
```

peut s'écrire également :

```
void init (int t[] )
{
int i;
   for (i=0; i<10; i++){
       t[i]=rand()%256;
       printf("t[%d]=%d\n",i,t[i]);
   }
   putchar('\n');
}
```

ou encore

```
void init (int*t[NB_ELEMENT])
{
int i;
   for (i=0; i<10; i++){
       t[i]=rand()%256;
       printf("t[%d]=%d\n",i,t[i]);
   }
   putchar('\n');
}
```

Dans les trois cas le paramètre pointeur de la fonction init() va recevoir au moment de l'appel l'adresse du tableau qui sera passé en argument.

```
int main()
{
int tab[10];

   init(tab) ;  // L'affectation implicite est la suivante :
                // init ( t = &tab[0]) ;
                // c'est équivalent à
                // init (t = tab) ;
   return 0;
}
```

Tableaux à plusieurs dimensions

Quelque soit le nombre des dimensions d'un tableau, seul la première dimension du tableau est convertie en pointeur. Par exemple s'il s'agit d'une matrice d'entiers, un tableau à deux dimensions, la matrice est convertie en pointeur de tableau d'entiers. Pour cette raison il n'est pas nécessaire de connaître la taille de la première dimension du tableau mais en revanche la taille des autres dimensions doit obligatoirement être spécifiée dans le cas de tableau statiques et non dynamique. Par exemple :

```
void init_matrice(int m[][10], int ty,int tx)
```

```
{
int x,y;
   for(y=0; y<ty; y++){
       for (x=0; x<tx; x++){
           m[y][x]=rand()%256;
           printf("%4d",m[y][x]);
       }
       putchar('\n');
   }
}

int main()
{
int mat[25][10]; // 25 lignes de 10 colonnes chacune
   init_matrice(mat,25,10);
   return 0;
}
```

le paramètre peut s'écrire également :

```
int (*m)[10]    // pointeur de tableau de 10 int
```

Entre les trois formes possible pour le pointeur en paramètre destiné à recevoir un tableau les deux dernières formes int t[] et int t[NB_ELEMENT] sont à privilégier et meilleures que int * t du point de vue du sens.

int t[] spécifie clairement avec les crochets que c'est un tableau qui est attendu en argument au moment de l'appel.

Par ailleurs int t[10] est équivalent à int*t ou ont t[ ] et la taille donnée au tableau ne sert à rien pour la machine. Vous pourriez passer un tableau d'une autre taille sans problème. Toutefois cette indication est utile à la compréhension du code pour rappeler dans la fonction que c'est un tableau de 10 int qui est attendu ce qui explique pourquoi la boucle va de 0 à 10.

Une autre solution est de passer explicitement en paramètre la taille du tableau ce qui permet d'utiliser la fonction pour des tableaux de n'importe quelle taille :

```
void init(int t[], int nb_element)
{
int i;
   for (i=0; i<nb_element; i++){
       t[i]=rand()%256;
       printf("t[%d]=%d\n",i,t[i]);
   }
   putchar('\n');
}
//-------------------------------
int main()
{
int t1[10];
int t2[20];

   init(t1, 10);       // appel pour initialiser t1
   init(t2, 20);       // appel pour initialiser t2
   return 0;
}
```

d. Modification des données via un passage par adresse

Qu'imprime selon vous le code suivant ?

```
void init(int t[], int nb_element)
{
int i;
   for (i=0; i<nb_element; i++)
       t[i]=1+rand()%9;
}

void affiche(int t[], int nb_element)
{
int i;
   for (i=0; i<nb_element; i++)
       printf("t[%d]=%d\n",i,t[i]);

   putchar('\n');
}

int main()
{
int tab[4]={0};

   affiche(tab,4);
   init(tab, 4);
   affiche(tab,4);
   return 0;
}
```

Le tableau tab est initialisé à la déclaration avec des 0. Le premier appel de la fonction affiche() affiche une liste de quatre 0 :

```
tab[0]=0
tab[1]=0
tab[2]=0
tab[3]=0
```

Le second appel affiche cette fois le tableau avec des valeurs initialisées de 1 à 9, j'ai obtenu :

```
tab[0]=6
tab[1]=9
tab[2]=8
tab[3]=5
```

Pourquoi le tableau tab qui est une variable locale au contexte d'appel est-il modifié après l'exécution de la fonction ? ...

Parce que la valeur transmise au paramètre int t[ ] de la fonction init() est une adresse mémoire. Lorsque la fonction reçoit le tableau tab en argument on a implicitement :

```
init(t=&tab[0], 4);
```

Ce qui veut dire que la variable pointeur t[ ] de la fonction init() contient ensuite la même adresse mémoire que celle du tableau tab déclaré dans le main(). Ainsi tout ce qui est fait sur t[ ] dans la fonction est fait sur tab dans le main() puisque c'est le même endroit dans la mémoire. C'est pourquoi tab est modifié après l'appel de la fonction init().

Grâce à l'utilisation d'adresse mémoire le tableau passé en paramètre peut être modifié dans la fonction. Voici une autre illustration de cette propriété :

Soit la fonction suivante :

```
void test (int t1[], int nb1, int t2[], int nb2)
{
   init(t1,nb1);
   init(t2,nb2);
   affiche(t1,nb1)
   affiche(t2,nb2);
   init(t1);
   affiche(t2);
}
```

et l'appel dans un main()

```
int main()
{
int tab[5];
   test(tab, 5, tab, 5);
   return 0;
}
```

Le tableau tab est passé en argument aux deux paramètres de la fonction test. A votre avis que se passera t-il ? ...

On peut traduire la situation dans la fonction test() de la façon suivante :

1) tab via son adresse est initialisé avec des valeurs aléatoires entre 1 et 9

2) tab via son adresse est initialisé avec des valeurs aléatoires entre 1 et 9

3) tab via son adresse est affiché

4) tab via son adresse est affiché

5) tab via son adresse est initialisé avec des valeurs aléatoires entre 1 et 9

6) tab via son adresse est affiché

Les variables t1[] et t2[] sont différentes mais comme c'est la même adresse mémoire qui est affectée à chacune au moment de l'appel, elles travaillent toutes les deux au même endroit de la mémoire c'est à dire sur le tableau tab.

3. Le retour d'un tableau statique est impossible

Il n'est pas possible d'utiliser le mécanisme de retour pour renvoyer un tableau statique qui serait déclaré dans la fonction. En tant que variable locale à la fonction le bloc mémoire qui lui est associé est libéré à la fin de la fonction et l'adresse mémoire du tableau n'est plus alors une adresse valide accessible en écriture. Cette adresse est transmissible au contexte d'appel mais écrire dedans provoque une sortie du programme, un plantage.(ce point est détaillé avec le module sur les pointeurs et l'allocation dynamique de tableaux.)

4. Quelques fonctions de traitement de chaines de caractères

a. Récupérer une chaine entrée par l'utilisateur

```
char* fgets (char*, int, FILE*)    -------   <stdio.h>
```

Cette fonction lit une chaîne entrée sur le fichier spécifié au paramètre p3 (dans le cas de l'entrée standard ce fichier sera stdin) jusqu'à un '\n' ou jusqu'à ce que p2-1 caractères aient été lus, le '\n' final compris. Tous les caractères lus sont placés à partir de l'adresse spécifiée au paramètre p1. Un \0 est ajouté comme dernier caractère après le \n.

```
int main(int argc, char *argv[])
{
char buf[100];
   printf("enter une chaine :\n");
   fgets(buf,100,stdin);
   printf("%s",buf);
   return 0;
}
```

b. Obtenir la taille d'une chaine

```
size_t     strlen(const char*s)   -------   <string.h>
```

Cette fonction retourne la longueur de la chaîne passée en paramètre sans compter le '\0' final (si p pointe sur une chaine vide " " strlen retourne 0). De ce fait la taille pour coder une chaîne complète est strlen(s)+1.

```
char buf[100];
int L;
   printf("enter chaine : ");
   fgets(buf,100,stdin);
   L=strlen(buf)
   printf ("longueur de la chaîne %s : %d\n",buf,L);
```

Attention, fgets() ajoute un '\n' lorsque l'on tape enter et ce caractère fait partie de la chaîne et compte pour un. Il est visible si l'on affiche la chaîne parce que le curseur d'écriture va passer à la ligne suivante.

c. Copier une chaine

```
char*      strcpy(char*, const char*)  -------   <string.h>
```

Cette fonction recopie la chaîne de caractères p2 à l'adresse p1 et retourne p1. Attention, la zone mémoire pointée par p1 doit être assez grande et correctement allouée, accessible en écriture.

```
char buf[100];
char cpi[100];
   printf("enter chaine :\n");
   fgets(buf,100,stdin);
   strcpy(cpi, buf);
   printf("copie : %s",cpi);
```

d. comparer deux chaines

```
int        strcmp(const char*, const char*)  -------   <string.h>
```

Cette fonction compare les chaînes de caractères p1 et p2 et retourne une valeur négative, nulle ou positive selon que la première est inférieure, égale ou supérieur à la seconde pour l'ordre lexicographique.

```
char buf[100];
char cpi[100];
int i;
   printf("enter deux chaines :\n");
   fgets(buf,100,stdin);
   buf[strlen(buf)-1]='\0';      // suppression  '\n' avant fin

   fgets(cpi,100,stdin);
   cpi[strlen(cpi)-1]='\0';      // suppression  '\n' avant fin

   if ( (res=strcmp(cpi,buf))==0)
       printf ("les chaines sont identiques\n");
   else if(res<0)
       printf("la chaine %s est avant la chaine %s\n",cpi,buf);
   else
       printf("la chaine %s est apres la chaine %s\n",cpi,buf);
```

e. Concaténer deux chaines

```
char*      strcat(char*, const char*)  -------   <string.h>
```

Cette fonction concatène la chaîne de caractères p2 à la suite de la chaîne p1 et retourne p1. La zone mémoire pointée par p1 doit être suffisamment grande et accessible en écriture.

```
char buf[100];
char cpi[100];
int i;
   printf("enter chaine :\n");
   fgets(buf,100,stdin);
   // suppression de '\n', avant dernier caractère
   buf[strlen(buf)-1]='\0';
   // la chaîne est mise trois fois à la suite dans cpi
   for (i=0; i<3;i++)
      strcat(cpi, buf);
```

```
printf("concat : %s\n",cpi);
```

5. Expérimentation : tableaux et fonctions

```
/*****************************************************************
TABLEAU ET FONCTION
- renvoie à la visibilité (accessibilité) des variables dans un
programme :

VARIABLES LOCALES :
- les variables sont locales à la fonction dans laquelle elles
sont déclarées. C'est à dire qu'elles sont visibles (accessibles)
uniquement dans le bloc de la fonction et dans tous ses sous blocs
imbriqués.

- Les valeurs des variables peuvent circuler grâce aux paramètres
   d'entrées et au mécanisme de retour (return)

VARIABLES GLOBALES :
- il est possible cependant de déclarer des variables au niveau
fichier en dehors de tout bloc, au dessus du main(). La variable
est alors accessible de tous les blocs et toutes les fonctions
```

```
dans le fichier, sans avoir à utiliser les paramètres ou le
return.

- cette propriété est utilisée pour les définitions de type (voir
les struct cours suivant) et des valeurs constantes (#define, enum
...)

- elle peut s'avérer utile pour simplifier l'écriture de petits
programmes de moins de 500 lignes de code. Elle ne s'applique
alors que sur quelques variables essentielles du programme.

- utilisée à mauvais essien cette propriété rique de nuire au
développement (en général le développement se trouve rapidement
paralysé au delà de 1000 lignes de code)

*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// valeur constante définie en globale
#define NBMAX  50

// tableau défini et déclaré en global, accessible de partout dans
// le prg
int tabGlo[NBMAX];

// déclaration des fonctions définies en dessous du main() (non
// encore définies au moment de l'appel)
void  init_tabGlobal    (void);
void  affich_tabGlobal  (void);

///*******************************************************
///*******************************************************
int main()
{
   init_tabGlobal();
   affich_tabGlobal();

   return 0;
}
///*******************************************************
///       la fonction ne peut agir que sur tabGlo
///*******************************************************
void init_tabGlobal()
{
int i;
   for (i=0; i<NBMAX; i++)
      tabGlo[i]=rand()%256;
}
///*******************************************************
///       la fonction ne peut agir que sur tabGlo
///*******************************************************
void affich_tabGlobal()
{
int i;
   for (i=0; i<NBMAX; i++){
      printf("%4d",tabGlo[i]);
      if (i%10==9)
         putchar('\n');
```

```
   }
}
/*****************************************************************
TABLEAU LOCAL EN PARAMETRE DE FONCTION

Quelle est la valeur numérique d'un tableau ?

- Soit par exemple la déclaration :

  int tab[10]; // Que vaut tab ?
               // Qu'est ce que tab pour la machine ?

tab c'est l'adresse mémoire du bloc alloué pour le tableau, c'est
à dire l'adresse du premier élément du tableau et ils sont tous
consécutifs. Passer un tableau à un paramètre de fonction ça
suppose passer une adresse mémoire au paramètre.
Le type de variable qui prend pour valeur des adresses mémoire est
le type pointeur.
Donc, pour passer un tableau à une fonction il lui faudra un
pointeur en paramètre

*****************************************************************/
/*
#define NBMAX_1   20
#define NBMAX_2   NBMAX_1*2

void     init        (int tab[], int taille);
void     affiche     (int tab[], int taille);

///*******************************************************
///*******************************************************
int main()
{
int tab[NBMAX_1]; // tableau local au main()

   printf("---------init et affiche tab :\n");
   init(tab, NBMAX_1);
   affiche(tab,NBMAX_1);

int tab2[NBMAX_2];
   printf("---------init et affiche tab2 :\n");
   init(tab2, NBMAX_2);
   affiche(tab2,NBMAX_2);

   return 0;
}
///*******************************************************
///   la fonction peut agir sur n'importe quel tableau
///   d'entiers.
///
///   IMPORTANT :
///   Parce que l'on écrit à une adresse mémoire les
///   modifications effectuées dans la fonction sont
///   répercutées sur la variable du contexte d'appel qui
///   a donné son adresse (ce que l'on appelle un passage
///   par rédérence ) :
///
///*******************************************************
void init(int tab[], int taille) // ou int *tab
{
```

```
int i;
   for (i=0; i<taille; i++)
      tab[i]=rand()%256;
}
///*******************************************************
///   la fonction peut agir sur n'importe quel tableau
///   d'entiers à une dimension
///*******************************************************
void affiche(int tab[], int taille) // ou int *tab
{
int i;

   for (i=0; i<taille; i++){
      printf("%4d",tab[i]);
      if (i%10==9)
         putchar('\n');
   }
}
*/
///*******************************************************
///
///   IMPORTANT :
///
///   Le retour d'un tableau statique local à une fonction
///   est impossible : comme pour toutes les variables, la
///   mémoire allouée par la machine est libérée à l'issue
///   de l'execution du bloc de la fonction.
///   Si l'on retourne l'adresse d'un tableau local
///   celle-ci n'est plus allouée dans la suite des opérations
///   et écrire à une adresse non réservée provoque un plantage
///   ou un comportement incertain du programme.
///
///   Pour que ça marche il faut allouer dynamiquement son
///   tableau, à savoir allouer soi-même la mémoire via un
///   pointeur et une fonction d'allocation (malloc(),calloc()
///   realloc() )
///
///
///*******************************************************
/*
int * init()
{
int tab[10];
int i;
   for (i=0; i<taille; i++)
      tab[i]=rand()%256;
   return tab;               // ERREUR!!!!!!!!!!!!!!!!!
}
*/
/
*****************************************************************
TABLEAU LOCAL STATIQUE DE 2 à n DIMENSIONS EN PARAMETRE DE
FONCTION :

- seule la première dimension est convertie en pointeur. Pour
toutes les autres il faut spécifier la taille

*****************************************************************/
/*
#define DIM1   10
```

```
#define DIM2   15

void     init        (int tab[][DIM2]);
void     affiche     (int tab[][DIM2]);

///*******************************************************
///*******************************************************
int main()
{
int dd[DIM1][DIM2]; // tableau 2D local au main()

   printf("---------init et affiche tab 2D :\n");
   init(dd);
   affiche(dd);

   return 0;
}
///*******************************************************
///*******************************************************
void init(int t[][DIM2])
{
int i,j;
   for (i=0; i<DIM1; i++)
      for (j=0; j<DIM2; j++)
         t[i][j]=rand()%256;
}
///*******************************************************
///*******************************************************
void affiche(int t[][DIM2])
{
int i,j;
   for (i=0; i<DIM1; i++){
      for (j=0; j<DIM2; j++)
         printf("%4d",t[i][j]);
      putchar('\n');
   }
}
*/
```

6. Mise en pratique : tableaux et fonctions

a. Appels de fonctions, tableaux en paramètre

Exercice 1
Ecrire une fonction qui affiche une lettre sur deux d'une chaine de caractères donnée en paramètre. Tester la fonction dans un programme

Exercice 2
Faire une fonction qui reconnaît et affiche des lettres communes à deux mots ou phrases saisies au clavier. La saisie est elle même une fonction. Tester dans un programme qui s'arrête à la demande de l'utilisateur.

Exercice 3
Faire une fonction qui compte le nombre de répétition des lettres dans un mot ou une phrase entré par l'utilisateur. Le résultat est affiché avec une fonction différente. Tester dans un programme qui s'arrête à la demande de l'utilisateur.

Exercice 4
Faire une fonction qui permet d'examiner la distribution des valeurs aléatoires obtenues avec la fonction rand() pour un nombre n de tirages entrés par l'utilisateur et sur 10 plages (des résultats de 0 à 9). Tester dans un programme qui s'arrête à la demande de l'utilisateur.

Exercice 5
Il existe une méthode de détermination de nombres premiers connue sous le nom de "crible d'Erastothène". Elle permet d'obtenir tous les nombres premiers inférieurs à une valeur n donnée. La méthode consiste à faire la liste de tous les nombres de 1 à n et à rayer successivement tous les multiples des entiers.

La méthode est :

1) rayer 1 qui n'est pas un nombre premier

2) passer au suivant non rayé (au départ 2) et rayer tous ces multiples (et uniquement ses multiples)

3) recommencer, chercher le suivant non rayé et rayer tous ses multiples, jusqu'à atteindre la fin de la liste

4) afficher la liste des nombres premiers.

Faire trois fonctions, une dans laquelle l'utilisateur spécifie le nombre n d'entiers dans la liste, une pour déterminer les nombres entiers dans la liste, une pour l'affichage de la liste. Le programme quitte si l'utilisateur le commande.

Exercice 6
Master mind. Faire un programme qui choisit au hasard une combinaison à cinq chiffres compris entre 1 et 8 et qui demande à l'utilisateur de la deviner. A chaque proposition de l'utilisateur le programme précise :
- □ le nombre de chiffres exactes ET à la bonne place
- □ le nombre de chiffres exactes PAS à la bonne place

Les propositions de l'utilisateur sont fournies sous la forme de cinq chiffres (pas de séparateur). L'entrée utilisateur doit être "blindée" c'est à dire traite les erreurs : entrée de lettres, réponse trop courte ou trop longue, chiffres incorrectes (non compris entre 1 et 8)

Un nombre maximum de coups est prévu pour chaque partie. Il est affiché et décroit à chaque à chaque proposition. Si tous les coups possibles sont passés l'utilisateur à perdu et le programme donne la réponse. Ensuite il propose une nouvelle partie.

Exercice 7
Afficher au hasard un certain nombre d'étoiles (*) dans un rectangle. Le nombre d'étoile et la taille du rectangle sont entrées par l'utilisateur sachant que la taille maximum est 25 par 80.

Attention, il ne peut pas y avoir d'étoiles superposées.

Exercice 8
Soit l'extrait de code suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <conio.c>

#define TX    30  // taille zone de jeu en x
#define TY    20  //          "         en y
#define DECX  5   // décalage par rapport au bord gauche
#define DECY  5   // décalage par rapport au bord haut

// la  contrée des terres du milieu
#define BORD   1   // dans la matrice signifie 1 bord
#define MUR    2   // dans la matrice signifie 1 mur

int player[2];   // les coordonnées du player 0 pourx et 1 pour y
int level[TY][TX]={
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,2,2,2,2,2,2,0,0,2,0,0,2,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,2,0,0,2,2,2,2,2,2,2,0,0,0,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,2,0,0,2,0,0,0,0,0,0,0,0,0,2,2,2,1,
1,0,0,2,2,2,2,2,2,2,0,0,2,0,0,0,0,0,0,0,2,0,0,0,0,0,0,1,
1,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,0,0,2,1,
1,0,0,2,2,2,2,2,2,2,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,2,2,2,2,2,2,2,2,0,0,2,0,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,1,
1,0,0,2,2,2,2,2,2,2,0,0,2,0,0,0,0,0,0,0,0,0,2,2,0,2,2,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
};
```

L'objectif est d'avoir un player qui se déplace au clavier dans le monde représenté par la matrice level. Pour ce faire écrire quatre fonctions :

- une fonction qui permet d'afficher à une position donnée, un caractère donné d'une couleur donnée
- une fonction d'initialisation du terrain à partir de la matrice level
- une fonction d'initialisation du player : il prend position dans son monde en respectant le terrain et ces murs
- une fonction qui gère les entrées clavier à savoir le déplacement du player et la fin du programme.

Exercice 9

Dans un jeu de démineur le terrain est spécifié par une matrice de nombres. Faire deux fonctions, une pour l'initialisation avec un nombre de mines répandues aléatoirement dans le terrain et une autre pour indiquer combien il y a de mines autour de chaque position dans la zone de jeu. Faire un programme de test qui affiche les mines en rouge et pour chaque position le nombre des mines à proximité s'il y en a (pas d'affichage sinon).

b. Manipulations sur les chaines

Exercice 10

Ecrire vos propres versions des fonctions suivantes :

- □ fgets() /saisir une chaine.
- □ strlen() / compter la longueur de la chaine et retourner le résultat
- □ strcpy() / copier une chaine dans une seconde
- □ strcat() / concaténer deux chaines, la seconde à la suite de la première
- □ strcmp() / comparer deux chaines et donner l'ordre lexicographique

Ensuite dans un programme, tester la différence de rapidité d'exécution entre votre version et celle de la librairie standard (donner les deux vitesses). Faites un menu qui permet de choisir la fonction à tester.

Exercice 11
La librairie standart <string.h> fournit d'autres fonctions pour le traitement des chaines. Par exemple les trois fonctions : strncat(), strncmp(), strncpy(). Trouver comment utiliser ces fonctions et dans un programme, donner vous-même un exemple de code qui marche pour chacune des trois.

Exercice 12
Faire une fonction qui compte le nombre de mots dans un texte (au choix, le texte est fourni en dur dans le programme ou une phrase assez longue peut être entrée par l'utilisateur). Tester dans un programme qui s'arrête à la demande de l'utilisateur. Faire une deuxième fonction qui compte le nombre de segments obtenus à partir de deux séparateurs quelconques entrés par l'utilisateur.

Exercice 13
Faire un programme avec un menu utilisateur qui permet les opérations suivantes tant que l'utilisateur le souhaite, chaque opération est une fonction :

- □ saisir une chaine
- □ compter le nombre de voyelles dans une chaine en paramètre et retourner le résultat.
- □ compter le nombre de consonnes et retourner le résultat
- □ supprimer toutes les consonnes d'une chaine et modifier la chaine en conséquence
- □ supprimer toutes les voyelles d'une chaine sans modifier l'originale
- □ inverser l'ordre des lettres de la chaine (bonjour -> ruojnob)
- □ indiquer si une chaine est un palimdrôme (abccba, anna, azertytreza sont des palindromes)
- □ crypter une chaine avec un décalage circulaire. La valeur de cryptage est entrée par l'utilisateur.
- □ décrypter la chaine si elle est cryptée.

Exercice 14
Dans un programme faire une fonction qui affiche la conjugaison au présent de l'indicatif d'un verbe du premier groupe sous la forme :

```
je chante
tu chantes
il/elle chante
nous chantons
vous chantez
ils/elles chantent
```

Le verbe à conjuguer est entré par l'utilisateur et passé en argument à la fonction.
Au départ s'assurer que le verbe fourni se termine bien par "er". On suppose qu'il s'agit d'un verbe régulier et que l'utilisateur ne fournit pas de verbe comme manger (cause nous mongeons et non nous mangons).

Exercice 15
Écrire une fonction qui compte le nombre de répétition des lettres utilisées dans un petit texte. La saisie du texte n'est pas faite dans cette fonction. Le résultat est affiché dans le contexte d'appel, après l'exécution de la fonction. Tester dans un programme.

Exercice 16
Problème mon clavier est cassé. A chaque fois que je tape sur s, j ou g ça écrit ch. Dans un programme tester une fonction qui remplace toutes les lettres s, j et g d'un petit texte passé en argument en ch : "suppose" devient "chuppoche", "je" devient "che", "fromage" devient "fromache" etc.

Ensuite intégrer dans le programme le fait que l'utilisateur puisse choisir lui-même la transformation à opérer. Faire une deuxième fonction qui prend en plus en argument la lettre à transformer et en quelle autre lettre ou ensemble de lettres elle est transformée.

H. Gestion des variables

1. Visibilité des variables

La visibilité et la durée de vie d'une variable est relative au lieu de sa déclaration dans le programme. Le niveau d'imbrication de bloc est appelé la profondeur de sa déclaration.

a. Profondeur de la déclaration

La déclaration est dite de profondeur 0 lorsqu'elle est en dehors de tout bloc d'instruction et de profondeur n avec n supérieur ou égal à 1 lorsqu'elle est dans un bloc ; n correspond au niveau d'imbrication du bloc concerné, par exemple :

```
#include<stdlib.h>            // niveau 0 en global, hors bloc
int x=0;
void test(int a);

int main()
{                             // bloc niveau 1
int i=0;
   {                          // bloc niveau 2
   int  y=9;
       {                      // bloc niveau 3
       int w=rand();
           test(w);
       }
   }
   test(i);
}

void test(int a)
{                             // autre bloc niveau 1
int b;
          ...// instructions de la fonction
}
```

Le paramètre int a de la fonction test() est considéré comme de niveau 1, local à la fonction test().

b. Portée des variables

Une variable quelque soit son type (char, short, int float double, struct, tableau, pointeur) est visible dans le bloc où elle est déclarée à partir de sa déclaration et dans tous les sous blocs (niveau d'imbrication supérieur). En revanche elle n'est pas accessible dans les blocs au-dessus (niveau d'imbrication inférieur) ou d'autres blocs séparés de même niveau. Si la variable est déclarée hors bloc c'est à dire au niveau 0 du fichier elle est visible dans tout le fichier à partir de sa déclaration. C'est ce qui est appelé une variable globale.

c. Masquage d'une variable

En C toutes les déclarations de variables sont regroupées en début de bloc pour chaque fonction. Il est rare de déclarer des variables dans des blocs imbriqués et les variables globales sont peu utilisées et très contrôlées (une variable est déclarée en globale pour une raison sérieuse) ceci afin d'éviter par exemple ce genre de problème :

```
#include <stdlib.h>
int x=1000;
int test(int x);

int main()
{
int x;
   x=1;
   printf ("%d, ",x);
   {
   int x;
       x=2;
       printf ("%d, ",x);
       {
       int x;
           x=3;
           printf ("%d, ",x);
       }
       {
       int x;
           x=33;
           printf ("%d, ",x);
       }
       printf ("%d, ",x);
   }
   x++;
   printf ("%d, ",x);
   x=test(x);
   printf("%d.",x);
   return 0;
}

int test(int x)
{
   x+=50;
   return x;
```

```
}
```

Qu'imprime le programme ci-dessus ? Les cinq variables du programme ont toutes le même nom mais il y a une priorité entre les variables : une déclaration de variable locale dans un bloc de niveau n, paramètres de fonctions compris, masque toutes les déclarations des variables de même nom des niveaux précédents.

Le programme ci-dessus imprime : 1, 2, 3, 33, 2, 2, 52.

La variable x déclarée en globale hors bloc avant le main() n'est pas touchée, elle se trouve dans cet exemple toujours masquée. Dans la fonction test() elle est masquée par le paramètre int x de la fonction. Les variables masquées ne sont pas touchées et conservent leur valeur. Elles redeviennent accessibles à la sortie du bloc quand elles ne sont plus masquées.

2. Durée de vie des variables

a. Variables globales

Une variable globale, c'est à dire déclarée hors bloc, existe pendant toute l'exécution du programme à partir de sa déclaration.

b. Variables locales (auto)

En revanche la durée de vie d'une variable locale à un bloc a la durée de vie du bloc. Elle dure dans le programme le temps du bloc où elle est déclarée. Par exemple :

```
   void tatatata()
   {
   int x, y ;
          // instructions
   }
```

Les variables x et y sont des variables dites automatiques pour lesquelles le mot clé "auto" est implicitement ajouté à la compilation. Elles apparaissent avec leurs déclarations à l'entrée d'un bloc et elles disparaissent automatiquement à sa sortie. Un espace de mémoire leur est attribué automatiquement à l'entrée du bloc et cet espace de mémoire est automatiquement libéré à la fin de l'exécution du bloc.

c. Variables static

Il existe cependant une autre classe d'allocation des variables, c'est la classe des variables dites static. Une variable déclarée static existe pendant toute la durée du programme dans le bloc où elle a été déclarée. Son emplacement mémoire n'est pas modifié. Si dans une fonction elle est initialisée à la déclaration, l'initialisation à lieu une seule fois, lors du premier appel de la fonction. Par exemple :

```
void  test( )
   {
   static int v=0;
          printf("toto=%d\n",v);
          v++;
```

```
   }
```

Chaque appel de cette fonction test() va afficher la valeur de la variable et ensuite l'incrémenter de 1. Dans le programme ci-dessous, la fonction est appelée à chaque pression sur une touche du clavier :

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

int main()
{
int fin=0;
   while (fin!='q'){
       if (kbhit()){
           fin=getch();
           test();
       }
   }
   return 0;
}
```

La première fois que l'utilisateur appuie sur une touche 0 est affiché ensuite 1, 2 3 etc.

Si la variable est globale le mot clé static sert à limiter sa portée au seul fichier de sa déclaration. Elle ne peut alors être visible que dans le fichier de sa déclaration et ne pourra en aucun cas être utilisée dans un autre fichier C du programme. Cette propriété est importante pour éviter toute confusion entre variables globales dans de gros programmes.

3. Choix méthodologiques

Le masquage, déclaration successive de variables de même nom à des niveaux différents, est évité en C parce que c'est une source de confusion qui nuit à la lisibilité d'un programme.

Les variables locales sont toujours définies en début de bloc, en général le niveau 1 à l'entrée du main et aux entrées des fonctions.

Une variable locale est déclarée static uniquement lorsque l'algorithme le justifie.

Pour les variables globales, l'important est d'avoir une raison sérieuse de choisir de mettre en global sa variable et ensuite de bien maîtriser son emploi. Elles sont toutes regroupées au début du fichier C qui les utilise et jamais disséminées dans le fichier. S'il y a plusieurs fichiers la variable globale est déclarée dans une librairie personnelle (ce point est abordé dans le chapitre suivant). Éventuellement elle peut être déclarée avec le mot clé "extern" dans les autres fichiers où elle est utilisée mais il est préférable de pouvoir s'en passer.

Dans des programmes de taille importante on peut limiter la zone d'influence d'une variable globale en la déclarant static à tel ou tel fichier.

4. Mise en pratique : gestion de variables

Exercice 1

De tête, quel résultat donne le programme suivant ?

```
#include<stdio.h>

int i=0;
int main()
{
int i=1;
   printf("i=%d\n",i);
   {
   int i=2;
       printf("i=%d\n",i);
       {
           i+=1;
           printf("i=%d\n",i);
       }
       printf("i=%d\n",i);
   }
   printf("i=%d\n",i);
}
```

Exercice 2

De tête, quel résultat donne le programme suivant ?

```
#include<stdio.h>

int f1(int);
void f2(void);

int n=10, q=2;

int main()
{
int n=0, p=5;
   n=f1(p);
   printf("A : dans main, n=%d, p=%d, q=%d\n",n,p,q);
   f2();
   return 0;
}

int f1(int p)
{
int q;
   q=2*p+n;
   printf("B : dans f1, n=%d, p=%d, q=%d\n",n,p,q);
   return q;
}

void f2(void)
{
int p=q*n;
   printf("C : dans f2, n=%d, p=%d, q=%d\n",n,p,q);

}
```

Exercice 3

De tête, quel résultat donne le programme suivant ?

```
#include<stdio.h>
#define LOW           0
```

```
#define HIGH          5
#define CHANGE  2

void workover(int i);
int reset(int i);
int i=LOW;

int main()
{
int i=HIGH;
   reset(i/2);
   printf("i=%d\n",i);
   reset (i=i/2);
   printf("i=%d\n",i);
   i=reset(i/2);
   printf("i=%d\n",i);

   workover(i);
   printf("i=%d\n",i);
}

void workover(int i)
{
   i=(i%i) * ((i*i)/(2*i) +4);
   printf("i=%d\n",i);
}

int reset (int i)
{
   if (i<=CHANGE)
       i=HIGH;
   else
       i=LOW;
   return i;
}
```

Exercice 4

Soit la fonction :

```
void wait(int tmps)
{
int start=clock();
   while (clock()<start+tmps){}
}
```

Modifiez cette fonction afin de déclencher une action tous les nb millisecondes mais sans arrêter le programme. Faire un programme de test.

Exercice 5

Faire deux fonctions différentes appelées dans une même boucle. Chaque fonction est ralentie avec une valeur différente et chacune affiche a chaque appel le nombre total de fois où elles a été appelée. Tester dans un programme.

Exercice 6

Soit un tableau d'entiers déclaré en global, faire deux fonctions d'initialisation (valeurs aléatoires) une avec paramètre, l'autre sans. Faire également deux fonctions d'affichage une avec paramètre et l'autre sans. Tester dans un programme.

I. Structuration d'un programme, étude d'un automate cellulaire

1. Clarifier et définir ses objectifs

Avec les tableaux les structures et les fonctions ils devient possible de faire des programmes assez importants éventuellement réparties sur plusieurs fichiers sources. Par exemple des versions simples de beaucoup de jeux classiques, pacman, casse brique, space invader, trétris etc. De tels programmes supposent une réflexion préalable : avec quelles variables et quelles fonctions et selon quel algorithme général je fais mon programme ? Mais dans le cas de programme que l'on imagine sur des sujets que l'on découvre, avant de pouvoir répondre définitivement à ces questions il est nécessaire de poser une hypothèse de départ et de se donner des objectifs à atteindre. Ces objectifs pourront évoluer au fur et à mesure du développement du projet et l'hypothèse de départ sera ou non corrigée en fonction des résultats obtenus.

L'idée ici est de faire un automate cellulaire 2D en mode console. Qu'est ce qu'un automate cellulaire ? Quel fonctionnement mettre en oeuvre ? Avec quelle structure de données ? Le premier point est de créer un espace de connaissance propre au projet sur lequel s'appuyer ensuite pour le réaliser.

a. Principe de l'automate cellulaire

Un Automate Cellulaire traduit le comportement d'un ensemble d'individus, de cellules, de points, etc. Il repose sur des règles simples exercées localement pour chaque individu en fonction de ses voisins immédiats. C'est le mouvement d'ensemble produit par toute la population soumise aux mêmes lois qui intéresse. Cette figure articule l'espace et le temps dans une évolution et elle n'est pas mathématisable parce que trop complexe. Le seul moyen de l'observer ou de l'utiliser est de la programmer pour un ordinateur. Pour cette raison c'est une figure emblématique de l'informatique. L'automate cellulaire se retrouve dans des domaines différents comme la modélisation de feux de forêt, l'évolution de population dans des villes, mais aussi le domaine graphique pour la richesse étonnante des figures qu'il peut générer.

Un chercheur de l'Université de Paris 8, Pierre Audibert le décrit ainsi : "A chaque étape de temps, tous ces individus évoluent en même temps, en fonction de leur situation locale. Le processus est typiquement parallèle (…) soumis à des règles simples dictées par leur voisinage immédiat, ils donnent lieu à des phénomènes complexes et contrastés. Apparitions localisées de formes organisées à partir d'un contexte aléatoire, ou inversement, développement de mouvements désordonnés et complexes qui cependant obéissent à des lois d'ensemble, partout règne l'harmonie des contraires. Cela concerne aussi bien des populations que l'évolution de formes dans le monde naturel, et même les incendies de forêt".

Comment à partir de ce texte faire un ou plusieurs programmes graphiques, juste pour voir ?

b. Fonctionnement envisagé

Nous allons nous appuyer sur un moteur d'organisation qui s'inscrit dans la tradition initiée dans les années soixante par John Horton Conway et nommée "Jeu de la vie".

Le principe est le suivant :

Chaque position dans le plan est dotée d'une valeur soit de zéro soit de un.

Au départ toutes les positions du plan sont mises à zéro et une fonction d'initialisation permet d'en mettre quelques-unes à un.

Ensuite toutes les positions du plan sont passées en revue. Pour chacune d'entre elles les positions voisines sont regardées et les valeurs trouvées comptabilisées, ce qui donne par exemple une situation comme :

(x-1, y-1) avec valeur 0	(x, y-1) avec valeur 1	(x+1, y-1) avec valeur 1
(x-1, y) avec valeur 1	(x, y) avec valeur 0	(x+1, y) avec valeur 0
(x-1, y+1) avec valeur 0	(x, y+1) avec valeur 0	(x+1, y+1) avec valeur 0

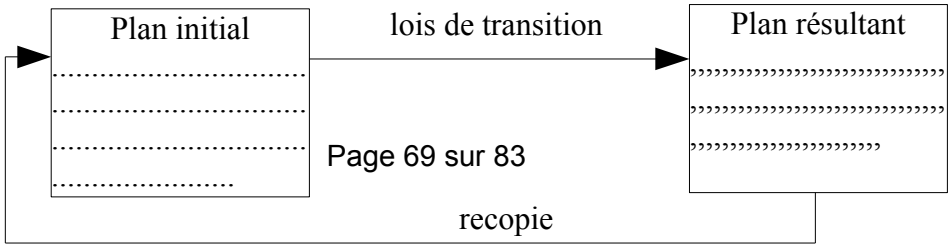
(x, y) au centre est la position courante. x est pour la position horizontale et y pour la position verticale. Il y a 3 positions adjacentes qui valent 1.

La valeur de la position courante va évoluer ou rester identique en fonction d'une règle de transition. La règle de transition repose sur les valeurs trouvées dans les positions voisines. Dans l'exemple ci-dessus, la position courante comptabilise trois 1 trouvés dans les cellules adjacentes. En quelque sorte il y a trois voisins. La règle de transition repose sur ce nombre de voisins.

Par exemple on aura une règle du type : si le nombre de voisins est inférieur à deux ou si le nombre de voisins est supérieur à trois, c'est-à-dire si le nombre de voisins est différent de 2 ou de 3, alors la position courante prend une valeur de un, sinon elle prend une valeur de zéro.

On peut imaginer d'autres règles, d'une façon générale le principe est : pour la position courante, si le nombre de voisins est inférieur à "un plancher" ou si le nombre de voisins est supérieur à "un plafond" alors la position courante prend la valeur booléenne "v1" sinon elle prend la valeur booléenne complémentaire "v2".

Il convient de préciser que le calcul se fait selon les positions du plan mais que les résultats, pour chaque position, sont stockés au fur et à mesure dans un plan équivalent en miroir du premier. Une fois que les valeurs de chaque position ont été recalculées et stockées dans le plan réservé aux résultats, ce dernier est recopié dans le plan initial et l'opération recommence :



2. Trouver une structure de données valable

Ok pour le principe de l'automate mais comment en faire un programme ? Comment traduire ce processus en une suite d'instructions dans un langage pour la machine ?

Une fois l'idée sur laquelle partir posée le premier point est de réfléchir à une possibilité de structure de données. Sur quoi va s'appuyer le moteur de l'automate pour fonctionner ? C'est à dire quel type de données je peux choisir pour coder mon automate ? variables simples ? Structure ? Tableaux ?

En l'occurrence ça saute aux yeux ici et ce n'est pas très compliqué. Essentiellement il y a deux surfaces 2D de booléens, des matrices matrices d'entiers feront très bien l'affaire, une pour le plan initial et une autre pour le plan résultant. La taille sera définie par deux macro constantes, ce qui donne :

```
#define TX  80
#define TY  50

int MAT[TY][TX];
int SAV[TY][TX];
```

3. Identifier les fonctions principales

Maintenant que nous savons à partir de quoi travailler, nos deux matrices, quelles sont les opérations à réaliser afin de mettre en œuvre le principe de fonctionnement que nous avons retenu ? C'est à dire quelles sont les opérations à faire et dans quel ordre ?

Il s'agit ici en s'appuyant sur nos deux matrices de repérer et d'imaginer un algorithme efficace pour le projet.

1) A priori la première étape est une étape d'initialisation :

- □ Au début les deux matrices sont initialisées à 0. Quelques positions du plan initial, la matrice MAT sont mises à 1. Faire une fonction d'initialisation.

2) Ensuite le moteur tourne en boucle et à chaque cycle il doit :

- □ Afficher le plan initial
- □ Calculer la transition pour chaque position et sauver le résultat dans SAV
- □ Recopier la matrice SAV dans le plan initial MAT

Il se dégage déjà quatre fonctions à faire :

- □ une fonction d'initialisation

- □ une fonction d'affichage du plan
- □ une fonction de calcul
- □ une fonction de recopie

Initialisation, affichage et recopie sont simples. La fonction de calcul nécessite un petit zoom :

Calculer c'est passer en revue (boucle) toutes les positions du plan initial et

- □ 1-pour chaque position compter le nombre des positions voisines à 1
- □ 2-appliquer la loi de transition en fonction de ce nombre
- □ 3-stocker le résultat dans la matrice SAV

Sur ces trois étapes dans l'algorithme, la première, le comptage des positions voisines peut faire l'objet d'une fonction à part. Cette fonction pourrait prendre en paramètre la position courante dans la matrice initiale et retourner le nombre de positions voisines à 1 trouvées tout autour.

La seconde étape pourrait aussi faire l'objet d'une fonction à part. Par exemple si le moteur était destiné à fonctionner selon différents paramétrages des lois de transition. Ces paramétrages pourraient alors être passés en paramètre.

Mais pour l'heure le moteur n'aura qu'une seul type de loi de transition. Le traitement dans la fonction calcul n'aura donc besoin que du comptage des positions voisines à 1 : une cinquième fonction, la fonction compte voisins.

4. Décider pour le niveau des variables fondamentales

Maintenant nous avons une hypothèse de structure de données et une hypothèse des fonctions à écrire, reste juste un point à décider : est ce que si les matrices sont des variables globales visibles de toutes les fonctions ou sont-elles encapsulées localement dans le main() ce qui nécessite une transmission via les paramètres ?

Quelle est la différence du point de vue du développement du projet ? Quels sont les avantages ou inconvénients de l'une et de l'autre ?

Si nos matrices sont globales il n'y a pas besoin de les passer en paramètre et ça allège un peu l'écriture des fonctions. C'est un petit programme qui tient sur un seul fichier, aucune confusion n'est possible avec d'autres processus à l'œuvre dans le même programme. Un problème pourrait éventuellement survenir ultérieurement si le développement du projet portait à faire tourner simultanément plusieurs automates et gérer plusieurs plans initiaux. Dans ce cas il serait préférable de pouvoir passer au moins le plan initial en paramètre.

Cependant il y aurait aussi la possibilité de partir sur un tableau de matrices déclaré en global et de modifier un peu chaque fonction en ajoutant une boucle pour NB matrices :

```
#define NB             10
int NBMAT[NB][TY][TX];   // 10 matrices de TY par TX
```

Mais à priori pour ce que nous voulons faire maintenant il n'y a pas d'obstacle à utiliser deux matrices en globale.

Les deux matrices et les macros sont déclarées en début de fichier après les include nécessaires et avant le main() :

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.c>

#define TX  80
#define TY  50

int MAT[TY][TX];
int SAV[TY][TX];
```

Remarque

En général toutes les variables globales sont spécifiées par des noms en majuscule, c'est une convention.

5. Écrire les fonctions

a. Fonction d'initialisation

La fonction d'initialisation est simple :

- mettre à 0 les deux matrices
- mettre à un quelques positions choisies

```
void init_matrice(void)
{
it i,j;

   // toutes les positions sont mises à 0
   for (j=0; j<TY; j++){
      for (i=0 ;i<TX ;i++){
         MAT[j][i]=0 ;
         SAV[j][i]=0;
      }
   }
         // sauf quatre positions au centre qui sont mises à 1
   MAT[TY/2][TX/2]=1;
   MAT[TY/2+1][TX/2]=1;
   MAT[TY/2][TX/2+1]=1;
   MAT[TY/2+1][TX/2+1]=1;
}
```

Le fait de mettre les deux matrices à 0 au départ va permettre d'appeler plusieurs fois l'initialisation des matrices pendant le fonctionnement du programme et de relancer le processus de propagation.

Voici la même fonction avec possibilité de transmission de différentes matrices de même taille (TY et TX) en paramètre :

```
void init_matrice(int M[][TX],int S[][TX])
{
int i,j;

   for (j=0; j<TY; j++){
```

```
      for (i=0 ;i<TX ;i++){
         M[j][i]=0 ;
         S[j][i]=0;
      }
   }
   M[TY/2][TX/2]=1;
   M[TY/2+1][TX/2]=1;
   M[TY/2][TX/2+1]=1;
   M[TY/2+1][TX/2+1]=1;
}
```

b. Fonction d'affichage

la fonction d'affichage utilise les fonction gotoxy() et textcolor (voir annexe fonctions console in-out). Le principe est simple parcourir la matrice et pour chaque position si la valeur est 1 colorer le fond en rouge, si la valeur est 0 colorer en bleu et dans les deux cas afficher un espace.

```
void affiche(void)
{
int x,y ;

   for (y=0; y<TY; y++){   // pour chaque position
      for (x=0 ;x<TX ;x++){
         gotoxy(x,y);       // déplacer le curseur à la position
         if(MAT[y][x]==1)  // si valeur 1
           textcolor(192); // couleur rouge en fond
         else
           textcolor(16);    // sinon couleur bleu en fond
         putchar(' '); // affiche un espace
      }
   }
}
```

c. Fonction de calcul

la fonction du calcul fait appelle à la fonction de comptage des voisins qui a besoin de la position courante en paramètre et retourne le nombre des voisins trouvés (les positions à 1). Pour chaque position la loi de transition est appliquée et donne 0 si les nombre de voisins est inférieur à 2 ou si le nombre de voisin est supérieur à 3 (si le nombre de voisin est différent de 2 et 3) et donne 1 dans tous les autres cas.

```
void calcul(void)
{
int x,y,nb_voisins;

   // pour chaque position dans la matrice (sans compter le
   // pourtour à cause de la recherche des voisins : de 1 à TY-1
   // et non de 0 à TY, idem pour x)
   for (y=1; y<TY-1; y++){
      for (x=1 ;x<TX-1 ;x++){

         // récupération du nombre de voisin à 1
         nb_voisins = compte_voisins(y,x);

         // application de la loi de transition basique et
         stockage du résultat dans la matrice SAV
```

```
         if (nb_voisins <2 || nb_voisins>3)
            SAV[y][x]=0;
         else
            SAV[y][x]=1;
      }
   }
}
```

La fonction calcul peut être un peu plus sophistiquée, prendre en compte la valeur de chaque position et varier la loi de transition :

```
// variante loi de transition :
  if (MAT[y][x]==1){ // par exemple prendre en compte l'état de la
                     // position courante

      if (nb_voisins <2 || nb_voisins>3 )//Possibilité également

         // de modifier 2 et 3
         SAV[y][x]=0;
      else
         SAV[y][x]=1;
   }
   if (MAT[y][x]==0){  // petite variation
      if (nb_voisins!=0 && (nb_voisins <2 || nb_voisins>3) )
         SAV[y][x]=1;
      else
         SAV[y][x]=0;
   }
```

d. Fonction compte voisin

A partir d'une position donnée retourner le nombre de positions voisines à 1 :

```
// en paramètre une position donnée
int compte_voisins(int y, int x)
{
int nb=0;                     // par défaut 0 voisin à 1

   if (MAT[y][x+1]==1) // examen de chaque position adjacente
     nb++;             // si 1, incrémenter le compte
   if (MAT[y-1][x+1]==1)  // les valeurs y-1, y+1, x-1, x+1
     nb++;                // doivent toujours rester dans la
   if (MAT[y-1][x]==1)    // matrice, attention aux débordements !
     nb++;
   if (MAT[y-1][x-1]==1)
     nb++;
   if (MAT[y][x-1]==1)
     nb++;
   if (MAT[y+1][x-1]==1)
     nb++;
   if (MAT[y+1][x]==1)
     nb++;
   if (MAT[y+1][x+1]==1)
     nb++;

   return nb;          // à la fin retourner le nombre trouvé
}
```

e. Fonction de recopie

Il y a deux possibilités de recopie. Copier via une boucle imbriquée ou utiliser une fonction de la librairie standart string.h probablement plus rapide et en une seule ligne.

```
void copie(void)
{
int i,j ;
   /*
   for (j=0; j<TY; j++){      // méthode basique
      for (i=0 ;i<TX ;i++){
         MAT[j][i]= SAV[j][i];
      }
   }*/
  // Avec la fonction memcpy dans string.h
  memcpy(MAT,SAV,sizeof(int)*TX*TY);
}
```

La fonction memcpy prend en paramètre p1 le tableau de destination, en p2 le tableau source et en p3 la taille en octet du bloc à copier. Le paramètre p1 doit être un espace mémoire suffisamment grand et correctement alloué.

f. Montage dans le main()

Reste plus qu'à monter la boucle principale du programme dans le main(). Il y a plusieurs possibilités. Nous avons choisi de faire avancer le programme à chaque fois qu'une touche est appuyée. Le programme prend fin si la touche q est appuyée.

```
int main(int argc, char *argv[])
{
int fin=0;

  printf("Action  : appuyer sur n'importe quelle touche\n"
         "Quitter : q");

  init_matrice();      // initialisation une fois au début

  while(fin!='q'){     // pendant la boucle principale

     if(kbhit()){      // si une touche est appuyée
        fin=getch();   // récup de la touche appuyée pour
                       // contrôler la fin
        affiche();     // le moteur d'organisation
        calcul();
        copie();
     }
  }
  return 0;
}
```

6. Intégrer une librairie personnelle

Dans la perspective de programmes importants en taille il est utile de transporter tout l'entête du fichier c'est à dire les include, les macros, les déclarations de variables globales, les définitions de type s'il y en a et les déclarations de fonctions dans une librairie personnelle. Une librairie personnelle est un fichier point h, nommé aussi fichier d'entête ou header en anglais. Ensuite tous les fichiers C du

programme qui utilisent les données déclarées dans la librairie feront un include de cette librairie.

Le programme que nous venons de présenter est un petit programme qui ne nécessite pas de librairie. Rien n'empêche toutefois de lui en adjoindre une. La démarche sera la même quelque soit la taille du programme.

Selon l'environnement de développement utilisé, créez un fichier point h si possible du même nom que le fichier.c auquel il correspond (ou par exemple le nom du programme s'il n'y a qu'un point h pour tous les fichiers c du programme). Ensuite retirer du fichier.c ce qu'il y a en en tête et mettez le dans le fichier.h :

```
// retiré de autocell.c et mis dans autocell.h

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.c>

#define TX  80
#define TY  50

int MAT[TY][TX];
int SAV[TY][TX];

void    init_matrice        (void);
int     compte_voisins      (int x, int y);
void    calcul              (void);
void    copie               (void);
void    affiche             (void);
```

en dans le fichier.c remplacer tout ça par :

```
#include "fichier.h"  // attention " " et non <  >
```

Attention !

Les signes < et > pour un include de librairie suppose que la librairie se trouve dans le dossier include du compilateur. En revanche les guillemets supposent que la librairie est dans le dossier du projet du programme.

7. Répartir son code sur plusieurs fichiers C

Notre programme ne nécessite pas une décomposition en plusieurs fichiers C mais nous allons le faire à titre d'exemple.

Décomposer son programme en plusieurs fichiers suppose une décomposition par unité de traitement. Il faut qu'il y ait un sens à cette décomposition, l'objectif est de s'y retrouver plus facilement dans le code. Pensez que vous ferez des programmes de plusieurs milliers de lignes, voire plusieurs dizaines ou centaines de milliers de lignes. A cette échelle les programmes deviennent comme des villes et il faut des plans très précis pour s'y retrouver. Il y a donc une architecture judicieuse à trouver dés qu'il s'agit de décomposer son code en fichiers. Pour notre programme par exemple nous pouvons mettre ce qui concerne le calcul sur un fichier, ce qui concerne l'affichage un autre et l'initialisation sur un dernier.

Cela rend déjà plus simple le développement du projet. Par exemple avoir plusieurs types d'affichage, plusieurs types de calcul et plusieurs initialisations possibles, le tout contrôlé par une interface dans le main().

Techniquement, pour avoir plusieurs fichiers il suffit de créer ces fichiers dans l'environnement de développement où l'on est et de les intégrer dans le projet. Ensuite d'y mettre le code voulu et l'include de sa ou de ses librairies.

Remarque

Lorsqu'il y a plusieurs fichiers.c avec plusieurs fois un include de la même librairie, afin que le compilateur ne fasse l'include qu'une seule fois, il est bon d'ajouter le jeu de directives suivant au début du header :

```
#ifndef FICHIER_H   // par convention reprendre le nom du
                    // fichier.h en majuscule
#define FICHIER_H

                    // mettre ses données ici
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.c>

#define TX  80
#define TY  50

int MAT[TY][TX];
int SAV[TY][TX];

void    init_matrice        (void);
int     compte_voisins      (int x, int y);
void    calcul              (void);
void    copie               (void);
void    affiche             (void);


#endif                 // pour clore
```

De la sorte l'inclusion n'aura lieu qu'une seule fois même si elle est spécifiée sur plusieurs fichiers.c.

et dans le fichier autocell.c

```
#include "autocell .h"
```

8. Mise en pratique : structuration d'un programme

a. Simulation d'un feu de forêt

En vous inspirant de l'automate cellulaire présenté, faire une simulation de feu de forêt. Une fois que votre espace représentant la forêt est constitué, le feu commence par exemple sur un bord et se propage en passant d'arbre en arbre lorsque des arbres se touchent. Il s'arrête sinon. Vous ne prendrez en compte que quatre directions : nord, est, sud, ouest. C'est un programme court quatre fonctions maximum.

b. Tristus et rigolus

Nous sommes dans un monde qui se partage entre les "tristus" aux personnalités tristes et négatives, voyant toujours ce qui va mal et réactivant sans cesse pour rien les problèmes insolubes et les "rigolus", toujours de bonne humeur, un peu légers, rigolant pour pas grand chose. Un tristus peut réussir à à attrister un rigolus et celui-ci finit par devenir un tristus s'il n'arrive plus à rigoler. Un rigolus peut réussir à faire progressivement rigoler un tristus qui se transforme alors en rigolus.
Faire la simulation d'un monde partagé entre les tristus (une couleur froide) et les rigolus (une couleur chaude). Nous verrons apparaître une ligne fluctuante de démarcation entre les deux sphères, ou plusieurs s'il y a plusieurs territoires...

c. Simulation d'une attaque de microbes dans le sang

Au microscope nous pouvons voir dans le sang d'un individu des microbes aux prises avec les globules du système immunitaire. Selon différents paramètres, la maladie s'étend ou ne s'étend pas. Faire une simulation simplifiée de ce processus en 2D.

d. Bancs de poissons, mouvements de populations

Trouver un moyen de simuler l'organisation collective du mouvement d'un ensemble de poissons …

e. Élection présidentielle

C'est la période avant le deuxième tour des élections présidentielles, chacun se demande pour qui il va voter entre les deux candidats.
Il y a :

- ceux qui prennent l'idée dominante de leur entourage proche
- ceux qui au contraire se révoltent contre les idées de leurs proches et prennent le contre pied de l'idée qui domine autour d'eux
- ceux qui sont indécis et adoptent de façon imprévisible soit l'un soit l'autre des points de vue
- ceux qui s'abstiennent parce qu'ils ne peuvent pas trancher entre des avis contradictoires d'égale intensité autour d'eux
- ceux qui s'abstiennent parce qu'ils ne veulent ni de l'un ni de l'autre de toute façon
- ceux qui restent fermes dans leurs convictions quelques soient les idées autour d'eux mais qui évoluent au fur et à mesure du temps

L'objectif est de faire une simulation graphique en 2D de l'évolution de l'opinion de la société jour après jour durant les trois mois qui précédent l'élection. Après avoir proposé une structure de données, établissez le principe algorithmique de votre moteur, listez les principales fonctions à écrire et écrivez et tester...

f. Chenille

Faire une chenille qui se déplace avec les flèches du clavier. Toujours le même processus de conception, tout d'abord discerner une structure de données valable :

que fait la chenille ? qu'est ce qui la distingue d'un player normal ? De quoi est-elle constituée ? Sous quelle forme la représenter et la coder ?
Ensuite repérer les grandes étapes du programme. Dans un premier temps faire peut-être une chenille simple avec une taille fixe. Ajouter ensuite un mécanisme qui la fait s'allonger. Ajouter encore un contrôle pour l'empêcher de se marcher dessus. C'est un programme court avec cinq fonctions maximum.

g. Système de vie artificielle, colonies de fourmis

Faire une simulation des interactions entre plusieurs colonies de fourmi. Les fourmis artificielles ont besoin de se nourrir et il y a au final NB colonies concurrentes de fourmis dans un monde unique. Faire un programme par étape. Le monde peut être ajouté dés le départ ou à n'importe quelle étape :

1) Fourmi seule
   - □ déterminer la structure de données pour une fourmi
   - □ écrire et tester une fonction d'initialisation pour une fourmi.
   - □ écrire et tester une fonction d'affichage / effacement pour une fourmi
   - □ écrire et tester une fonction pour faire bouger une fourmi

2) Une colonie de fourmis
   - □ déterminer la structure de données pour une colonie de NB fourmis
   - □ écrire et tester une fonction d'initialisation pour toute la colonie
   - □ écrire une fonction pour le mouvement de toute la colonie
   - □ écrire et tester une fonction d'affichage/effacement de la colonie
   - □ imaginez un type de relation entre les fourmis qui influe par exemple sur le mouvement de la colonie

3) Plusieurs colonies de fourmis
   - □ déterminer la structure de données NB colonies de fourmis
   - □ écrire et tester une fonction d'initialisation pour toutes les colonies
   - □ écrire une fonction pour le mouvement de toute les colonies
   - □ écrire et tester une fonction d'affichage/effacement des colonies
   - □ écrire une ou plusieurs fonctions pour gérer les relations des colonies

4) Ajouter un monde sur lequel les fourmis progressent
   - □ déterminer la structure de données pour le monde des fourmis
   - □ Initialiser et afficher le monde des fourmis
   - □ écrire une ou plusieurs fonctions pour gérer la relation d'une fourmi au monde
   - □ écrire une ou plusieurs fonctions pour gérer les relations de toutes les fourmis au monde.

h. Boutons et pages

Il s'agit de créer un système pour la gestion de ses boutons dans un programme console. Faire ce programme sur papier. Toutefois il est possible de faire ce programme sur machine en utilisant la librairie creaco fournie en téléchargement.

Un bouton est défini par une position, deux couleurs, un ou deux textes (tableaux de char). Il peut y avoir plusieurs "pages" dans votre programme. Le nombre des boutons et des pages est choisi par vous pour tester. Le changement des pages se fait en utilisant les flèches gauche et droite :

```
if (key_pressed(VK_LEFT))
   change_page_courante();                // vers page de gauche
if (key_pressed(VK_RIGHT))
   change_page_courante();                // vers page de droite
```

La fonction change_page( ) est donnée. Elle incrémente ou décrémente une variable globale : int page_courante; et elle se charge aussi de l'affichage à l'écran de la page sélectionnée.

Vous disposez également de la souris de la façon suivante :

```
la variable mouse_x donne sa position horizontale
la variable mouse_y donne sa position verticale
```

La fonction clic_pressed(int numClic) indique si clic, par exemple :

```
if (clic_pressed( 1 )==1)
    printf("clic gauche appuyé\n");
if (clic_pressed( 2 )==1)
    printf("clic droit appuyé\n");
```

Lorsqu'un bouton est cliqué vous appellerez une fonction action(boutonNB) selon le numéro du bouton activé.

1) Traiter un élément seul
   - □ structure de données, initialisation, affichage, action.

2) Traiter un ensemble d'éléments :
   - □ structure de données, initialisation, affichage, action, en utilisant les fonctions faites pour un éléments seul.

3) Traiter plusieurs ensembles d'éléments :
   - □ structure de données, initialisation, affichage, action, en utilisant les fonctions faites pour un ensemble d'éléments

i. Panneaux de bois et entrepôts

Une scierie possède plusieurs entrepôts dans lesquels elle stocke ses lots de panneaux de bois. Le nombre d'entrepôts et le nombre de panneaux sont choisis par vous (par exemple quatre entrepôts : nord, est, sud, ouest et 10000 panneaux max par entrepôt).

Un panneaux est défini par une longueur, une épaisseur, une largeur et un type de bois (quatre variétés : sapin, frêne, olivier, chêne).

Faire une base de données simple en console pour la gestion du stockage de tous les panneaux. Le projet peut se faire sur papier.

1) Traiter un élément seul
   - □ structure de données, initialisation, affichage, action.

2) Traiter un ensemble d'éléments :
   - □ structure de données, initialisation, affichage, action, en utilisant les fonctions faites pour un éléments seul.

3) Traiter plusieurs ensembles d'éléments :
   - □ structure de données, initialisation, affichage, action, en utilisant les fonctions faites pour un ensemble d'éléments

j. Nenuphs et clans

Dans un monde microscopique découvert par hasard sur une météorite tombée dans le Sahara vivent des étranges créatures fluorescentes en forme de petits nénuphars surnommées "Nénuphs". Le nénuph est mobile et coloré. Il vit à la surface des pierres dures et il n'a que deux dimensions. La nuit il produit une sorte de musique quantique inaudible pour nous. Le nénuph vit en clan et il peut y avoir plusieurs clans sur une même pierre. Selon une théorie récente les nénuphs et leurs musiques agissent sur l'infiniment petit et seraient à l'origine de la création de pierres précieuses...

L'objectif est de faire une simulation simplifiée en fenêtre console de la vie des nénuphs. Pour commencer pouvoir visualiser plusieurs clans.

1) Traiter un élément seul
   - □ structure de données, initialisation, affichage, action.

2) Traiter un ensemble d'éléments :
   - □ structure de données, initialisation, affichage, action, en utilisant les fonctions faites pour un éléments seul.

3) Traiter plusieurs ensembles d'éléments :
   - □ structure de données, initialisation, affichage, action, en utilisant les fonctions faites pour un ensemble d'éléments

k. Neige 1

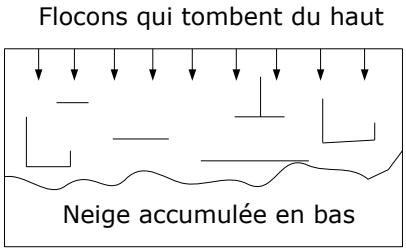
Neige 1. Faire un programme de flocons de neige qui tombent du haut de la fenêtre console en bas. Éventuellement les flocons peuvent afficher un message en descendant, par exemple "Bonne Année !" si c'est la bonne période.
Le premier point est de définir la structure de données : que fait un flocon ? comment peut-il apparaître dans notre environnement texte ? En un mot sous quelle forme, avec quoi coder un flocon ? Tous les flocons ?
Ensuite il faut repérer les grandes étapes du programme. C'est un programme court. Il y a essentiellement trois fonctions à faire et à appeler dans le main().

l. Neige 2

Faire un programme ou des flocons de neige tombent de haut en bas de l'écran mais à la différence du projet 1 les flocons s'accumulent en bas. Le problème est qu'à un moment donné il n'y aura plus de place dans la zone de jeu remplie de flocons. Trouver une solution pour gérer cette accumulation afin que la neige puisse tomber sans jamais s'arrêter.

m.Neige 3

Toujours sur le principe des flocons de neige qui tombent, il y a des obstacles sur le parcourt des flocons. Ils sont obligés soit de s'arrêter, soit de trouver la sortie lorsque c'est possible afin d'atteindre le sol :



n. Casse-brique base

Faire un casse brique simple. Le lanceur correspond à une seule case en bas de l'écran (ou sur un bord). Il se déplace avec des touches. Une touche permet d'envoyer des balles. Si une brique est touchée par une balle elle disparait. La balle ne rebondit pas. A chaque tir le lanceur envoie une nouvelle balle. Le but est de faire disparaître toutes les briques ou lettres et un message de succès s'affiche à la fin.

o. Casse-brique guru

Casse brique plus complet. Le lanceur a une dimension, par exemple trois cases. une balle lancée rebondie contre les obstacles qu'elle rencontre. Lorsqu'elle touche une brique la brique disparaît. Le lanceur doit la rattraper pour qu'elle reparte sinon cette balle est perdue. La partie est gagnée si toutes les briques sont détruites avec le nombre de balles disponibles.

p. Space invader base

Space invader simple. Les ennemis sont immobiles. Le player se déplace horizontalement dans le bas de l'écran et tire sur les ennemis. Un ennemi touché disparaît.

q. Space invader strong

Space invader more. Les ennemis sont toujours immobiles mais ils ont des propriétés de défense et d'attaque. Quelques uns mettent du temps avant de diparaître, ils doivent être touchés plusieurs fois. D'autres peuvent envoyer des boulets.

r. Space invader very strong

Space invader guru. Les ennemis sont mobiles latéralement et avancent petit à petit sur le player...

s. Pacman débutant

Pacman I. Un player avance dans un monde.

t. Pacman intermediate

Pacman II. Le monde où se trouve le player est parcouru par des ennemis. Il n'y a pas d'interaction entre eux mais ils ne peuvent pas passer par dessus les uns des autres. Si un ennemi rencontre le player il part dans une autre direction. De même si deux ennemis se rencontrent.

u. Pacman guru

Pacman III. Le player est armé et peut éliminer des ennemis au fur et à mesure qu'il les trouve. Éventuellement quelques ennemis peuvent riposter.

v. jeu de miroirs

Il s'agit de poser des miroirs sur un terrain afin ensuite de faire faire un parcourt à un rayon laser. A chaque fois que l'utilisateur tape "enter" un petit miroir apparaît et avec les flèches il est conduit à la place souhaitée. Lorsqu'un réseau est constitué, si l'utilisateur tape space, un lanceur apparaît sur un bord de l'écran ou ailleurs. Le joueur positionne le lanceur face au premier miroir et tire. Si le rayon réussit à parcourir tout le circuit c'est gagné. Il y a plusieurs façon d'aborder le scénario de ce programme.

w. Simulations football

L'objectif est d'entrer dans la programmation d'un jeu pour jouer au foot. Plusieurs niveaux et étapes sont à envisager :

- un joueur et un ballon
- deux joueurs et un ballon
- un joueur contre l'ordinateur
- un joueur et une équipe
- un joueur et une équipe et un ballon
- deux joueurs et une équipe et un ballon
- un joueur et son équipe contre l'ordinateur

Avancer progressivement en choisissant vos étapes selon votre niveau.