# TD/TME Séance 2
# Listes chaînées et simulation d'un écosystème - partie 1

Version du 27 août 2024

## Objectif(s)

- ✯ tableaux 2D statiques en arguments
- ✯ Listes chaînées (rappels)
- ✯ assert
- ✯ Makefile (bis)
- ✯ Bonnes pratiques de programmation : les tests
- ✯ Bonnes pratiques de programmation : ddd/gdb et valgrind

L'objectif de ce sujet est la réalisation d'un projet de programmation d'un écosystème virtuel utilisant des listes chaînées. Nous allons d'abord commencer par quelques exercices de rappel, puis nous aborderons le problème de l'écosystème.

# TD : des tableaux aux listes

### Exercice 1 *(obligatoire)* – Tableaux 2D, statiques et dynamiques

1. Soit le `main` suivant

```
#define DIM1 5
#define DIM2 6

int main(void) {

   char tab2D[DIM1][DIM2];

   InitTab(tab2D);

   return 0;
}
```

   La fonction `InitTab` permet de remplir le tableau de 0. Ecrire cette fonction. Attention au prototype...

2. Transformez ce programme en remplaçant le tableau statique par un tableau dynamique. Attention aux fuites mémoires. L'initialisation du tableau dynamique se fera au niveau du `main` et `InitTab` ne fera toujours que le remplissage de 0. Son prototype pourra être modifié.
3. Quels sont les avantages et inconvénients des tableaux statiques et dynamiques ? Dans quels cas utiliser l'un ou l'autre ?

### Exercice 2 *(base)* – Rappels sur les listes chaînées, analyse d'un exemple

Soit la structure de données suivante :

```
typedef struct _elt Elt;
struct _elt{
  int donnee;
  Elt *suivant;
};
```

Soit le programme suivant :

```
int main() {
  int taille=10;
  Elt *liste=NULL;
  Elt *nelt=NULL;
  int i=0;

  for (i=0;i<taille;i++) {
    nelt=malloc(sizeof(Elt));
    if (nelt == NULL) {
      printf("Erreur lors de l'allocation.\n");
      return 0;
    }
    nelt->donnee=i;
    nelt->suivant=liste;
    liste=nelt;
  }

  nelt=liste;
  while (nelt) {
    printf("%d ",nelt->donnee);
    nelt=nelt->suivant;
  }
  printf("\n");

  return 0;
}
```

1. Que fait ce programme ? Qu'est-ce qui est affiché à l'écran ?
2. Quelle est la place mémoire occupée par la liste chaînée créée dans ce programme ? Quelle taille ferait un tableau contenant les mêmes données ?
3. La mémoire allouée pour cette liste n'a pas été libérée. Ajoutez des instructions permettant de libérer toute la mémoire qui a été allouée.

# TD : Ecosystème - Mise en place

L'objet de cette partie du sujet est d'écrire des fonctions de manipulation de listes chaînées et de les utiliser pour programmer une simulation simple d'écosystème.

Le modèle d'écosystème que vous allez programmer n'a aucune prétention à être réaliste, mais il permet de se familiariser avec le concept d'équilibre, primordial dans un écosystème. Cet écosystème contiendra deux types d'entités virtuelles : des proies et des prédateurs, susceptibles de manger les proies. Notre écosystème est un monde discret (un tore, que nous afficherons comme un rectangle) contenant un certain nombre de cases, identifiées par leurs coordonnées (entières) x et y. Chaque proie (et chaque prédateur) est dans une case donnée et peut se déplacer. A un instant donné, une case peut contenir plusieurs proies et plusieurs prédateurs. Chaque case peut aussi contenir de l'herbe, la nourriture des proies.

La simulation de notre écosystème repose sur plusieurs structures de données et sur des fonctions que vous allez écrire dans la suite de cette séance. Les données utilisées pour la simulation sont une liste chaînée contenant les proies et une autre liste chaînée contenant les prédateurs ainsi qu'un tableau statique à deux dimensions représentant le monde.

## Exercice 3 *(obligatoire)* – Organisation du programme et compilation séparée

Le programme est organisé en quatre fichiers :
— `main_tests.c`, qui contiendra un main avec les tests de vos fonctions,
— `main_ecosys.c`, qui contiendra un main permettant de simuler un écosystème,
— `ecosys.c` qui contient toutes les autres fonctions de manipulation de listes.
— `ecosys.h` qui contient les prototypes des fonctions de ecosys.c et les définitions de structure (.h).

1. A votre avis, dans quels fichiers .c le fichier `ecosys.h` sera-t-il inclus ?
2. Donnez les lignes de commande permettant de compiler séparément chaque fichier .c puis de créer l'exécutable `test_ecosys` à partir de `main_test.o` et `ecosys.o` et l'exécutable `ecosys` à partir de `main_ecosys.o` et `ecosys.o`
3. Le fichier `main_test.c` a été modifié. Quelles commandes sont nécessaires pour que les exécutables soient à jour ?
4. Le fichier `ecosys.c` a été modifié. Quelle commandes sont nécessaires pour que les exécutables soient à jour ?
5. Ces opérations fastidieuses de recompilation sont grandement facilitées par l'utilisation de l'utilitaire make. Ecrivez le `Makefile` permettant de compiler les programmes `tests_ecosys` et `ecosys`.

## Exercice 4 *(obligatoire)* – Structure de données

Les proies et les prédateurs seront représentés par une même structure de données contenant les coordonnées entières `x` et `y`, un nombre réel `energie` (tel qu'un animal dont l'énergie tombe en dessous de 0 est mort, ce point sera géré plus tard). Il contiendra ensuite un tableau `dir` de deux entiers qui représentera sa direction (nous y reviendrons également plus tard). Comme nous souhaitons stocker des variables de ce type dans des listes simplement chaînées, la structure contiendra enfin un pointeur nommé `suivant` sur cette même structure.

Ces déclarations seront réalisées dans `ecosys.h`, de même que les prototypes des fonctions que vous écrirez par la suite. Les fonctions elles-mêmes seront écrites dans un fichier `ecosys.c`.

1. Ecrivez la déclaration de cette structure, à laquelle vous donnerez, via un `typedef` le nom équivalent `Animal`.
2. Ecrivez la fonction de création d'un élément de type `Animal` (allocation dynamique). Vous initialiserez les champs `x`, `y`, et `energie` à partir des arguments de la fonction. Les cases du tableau `dir` seront initialisées aléatoirement avec les valeurs -1, 0 ou 1.

   La fonction aura le prototype suivant :

   ```
   Animal *creer_animal(int x, int y, float energie);
   ```

   Pour que votre programme soit robuste et éviter les erreurs de segmentation lors du développement du programme, nous utiliserons des appels à `assert(expression);` qui (comme en python) arrêtent le programme lorsque l'expression est fausse. Par exemple, dans cette fonction, il faut vérifier que le pointeur retourné par `malloc` n'est pas `NULL`, ce que vous ferrez avec un appel à `assert`.
3. Ecrivez la fonction d'ajout en tête dans la liste chaînée.

   La fonction aura le prototype suivant :

   ```
   Animal *ajouter_en_tete_animal(Animal *liste, Animal *animal);
   ```

   N'oubliez par d'utiliser `assert` pour éviter les erreurs de segmentation potentielles.
4. Ecrivez des fonctions permettant de compter le nombre d'éléments contenus dans une liste chaînée. Vous écrirez une fonction itérative et une fonction récursive.

   Les fonctions auront les prototypes suivants :

   ```
   unsigned int compte_animal_rec(Animal *la);
   unsigned int compte_animal_it(Animal *la);
   ```

### Exercice 5 *(entraînement)* – Affichage et main

1. La fonction d'affichage de votre ecosystème affichera le contenu des différentes cases de votre monde simulé. Vous afficherez un espace pour les cases vides, une étoile ('*') pour les cases contenant au moins une proie, un 'O' pour les cases contenant au moins un prédateur et si une case contient des proies et des prédateurs, vous afficherez un '@'.

```
Nb proies (*):        20
Nb predateurs (O):    20
+----------+
| *        |
|* *O * **O|
| O*O      |
|  O  O    |
|  @  O  O |
|  **    *O|
|     @ O @|
|    *   O@|
|  O *O    |
|  O O @*  |
+----------+
```

   Pour cela vous déclarerez un tableau statique de **char** à 2 dimensions de taille `SIZE_X*SIZE_Y` et commencerez par le remplir d'espace. Ensuite, vous parcourrez la liste des proies et mettrez une étoile dans la case où les proies se trouvent, puis ferez de même avec les prédateurs. Enfin, vous afficherez ce tableau de caractères pour obtenir l'affichage précédent.

   L'herbe mentionnée en introduction n'est pas représentée visuellement ici. Vous pourrez l'ajouter pendant le TME.

# TME : Ecosystème - Simulation

### Exercice 6 *(obligatoire)* – Tests des fonctions et main

Les fonctions évoquées pendant le TD sont fournies, ainsi qu'un main permettant de tester ces fonctions (`main_tests.c`). Un Makefile est également fourni pour créer un exécutable `tests_ecosys`.

1. Il semblerait que les fonctions fournies contiennent des erreurs... Testez le programme, vérifiez si son comportement est correct et corrigez les erreurs éventuelles en vous appuyant sur les outils de débogage qui vous ont été présentés. Vous pouvez également ajouter des `assert` dans les fonctions pour vérifier que les variables ne sortent pas des valeurs ou intervalles attendus. Une bonne pratique consiste à insérer des `assert` chaque fois qu'un bug est découvert pour éviter qu'il ne se reproduise lors des évolutions ultérieures de votre code.
2. Pour faciliter les manipulation des listes de proies et de prédateurs, écrivez une fonction permettant d'ajouter un animal à la position `x`, `y`. Cet animal sera ajouté à la liste chaînée `liste_animal`. Plutôt que de renvoyer l'adresse du premier élément de la liste, vous passerez ce pointeur par adresse de façon à pouvoir le modifier directement dans la fonction : l'argument sera donc un pointeur sur liste chaînée, donc un pointeur sur un pointeur sur un Animal... Vous vérifierez (avec `assert`) que les coordonnées indiquées sont correctes c'est-à-dire positives et inférieures à `SIZE_X` ou `SIZE_Y` qui sont des étiquettes (**#define**) définies par ailleurs.

   La fonction aura le prototype suivant :

```
void ajouter_animal(int x, int y, Animal **liste_animal);
```

3. Ecrivez une fonction main dans un fichier `main_tests2.c` qui va créer 20 proies et 20 prédateurs à des positions aléatoires, vérifiez leur nombre en faisant appel aux fonctions de comptage vues précédemment et enfin affichez l'état de votre écosystème.

   Vous complèterez aussi le `Makefile` pour créer le programme `tests_ecosys2`.

4. Ecrivez la fonction `liberer_liste_animaux(Animal *liste)` qui permet de libérer la mémoire allouée pour la liste `liste`.

5. Complétez la fonction `main` de `main_tests2.c` pour libérer toute la mémoire allouée.

6. Pour vérifier qu'il n'y a pas de fuite mémoire, il faut utiliser l'utilitaire valgrind. Lancez votre programme avec lui (en tapant `valgrind` puis votre programme - ex : `valgrind ./tests_ecosys2`). L'objectif est de n'avoir aucun octet qui soit perdu :

   ```
   ==66002== LEAK SUMMARY:
   ==66002==    definitely lost: 0 bytes in 0 blocks
   ==66002==    indirectly lost: 0 bytes in 0 blocks
   ```

7. Ecrivez la fonction permettant d'enlever un élément de la liste chaînée et de libérer la mémoire associée. Comme précédemment, la liste sera passée par adresse.

   La fonction aura le prototype suivant :

   ```
   void enlever_animal(Animal **liste, Animal *animal);
   ```

   Ajoutez dans votre main la suppression de quelques proies et quelques prédateurs avant la libération de la liste entière, tout en vérifiant que les comptages sont toujours corrects. Vérifiez à nouveau qu'il n'y a pas de fuite mémoire avec valgrind.

# TD/TME Séance 3
# Listes chaînées et simulation d'un écosystème - partie 2

Version du 27 août 2024

## Objectif(s)

- ✯ lecture et écriture de fichiers
- ✯ débogage et gestion de la mémoire

# TD

### Exercice 1 *(obligatoire)* – Lecture/écriture de fichiers

Nous allons écrire l'ensemble de l'écosystème dans un fichier pour pouvoir le relire ultérieurement et reprendre l'observation de ses évolutions. Cela peut également aider à déboguer son fonctionnement en comparant l'état de l'écosystème avant et après une mise à jour, par exemple.

Voilà le format du fichier sur un exemple contenant 4 proies et 3 prédateurs :

```
<proies>
x=8 y=43 dir=[0 1] e=10.000000
x=18 y=38 dir=[1 -1] e=10.000000
x=9 y=37 dir=[-1 1] e=10.000000
x=18 y=0 dir=[0 1] e=10.000000
</proies>
<predateurs>
x=14 y=32 dir=[0 -1] e=10.000000
x=13 y=40 dir=[-1 -1] e=10.000000
x=18 y=46 dir=[0 -1] e=10.000000
</predateurs>
```

1. Ecrivez la fonction `ecrire_ecosys` permettant d'écrire ce fichier à partir des listes de proies et de prédateurs. Prototype :

   ```
   void ecrire_ecosys(const char *nom_fichier, Animal *liste_predateur, Animal *liste_proie);
   ```

   La fonction va écrire la ligne `<proies>`, puis toutes les proies, puis `</proies>`, puis `<predateurs>`, puis tous les prédateurs, puis elle termine pas la ligne `</predateurs>` avant de fermer le fichier.

2. Ecrivez la fonction `lire_ecosys` permettant de lire ce fichier et qui retourne les listes de proies et de prédateurs qui ont été lues. Cette fonction prendra en argument le nom du fichier à lire ainsi que les listes de proies et de prédateur par pointeur pour pouvoir les modifier. Ces variables seront supposées pointer sur des listes vides. Prototype :

   ```
   void lire_ecosys(const char *nom_fichier, Animal **liste_predateur, Animal **liste_proie);
   ```

La fonction devra lire ligne par ligne et construire les animaux à partir des éléments lus en les ajoutant à la bonne liste chaînée (proies ou prédateurs).

Nous allons maintenant simuler notre écosytème. Il fonctionne en temps discret. A chaque pas de temps un certain nombre d'opérations devront être réalisées :

— toutes les proies se déplacent, éventuellement changent de direction de déplacement, leur énergie est décrémentée de 1 ;
— si de l'herbe est disponible, une proie regagne autant d'énergie qu'il y a d'herbe sur la case ;
— les proies sont susceptibles de se reproduire avec une probabilité `p_reproduce_proie` ;
— tous les prédateurs se déplacent, éventuellement changent de direction de déplacement, leur énergie est décrémentée de 1 ;
— les prédateurs qui sont sur la même case qu'une proie la mangent. Dans ce cas la proie meurt et le prédateur augmente son énergie d'un montant valant l'énergie de la proie ;
— les prédateurs sont susceptibles de se reproduire avec une probabilité `p_reproduce_predateur`.

Les différentes probabilités évoquées ci-dessus seront des variables globales que vous déclarerez dans `ecosys.c`. Vous les déclarerez en tant qu'`extern` dans `ecosys.h`. Voici leur liste, avec des exemples de valeurs possibles :

```
 /* Parametres globaux de l'ecosysteme (externes dans le ecosys.h)*/
float p_ch_dir=0.01; //probabilite de changer de direction de deplacement
float p_reproduce_proie=0.4;
float p_reproduce_predateur=0.5;
int temps_repousse_herbe=-15;
```

Nous allons à présent détailler les différentes fonctions permettant de programmer cette simulation.

## Exercice 2 *(obligatoire)* – Déplacement et reproduction

Les mouvements seront gérés de la façon suivante : chaque proie (ou prédateur) dispose d'une direction indiquée dans le champ `dir`, qui est un tableau de deux entiers. Il indique la direction suivie sous la forme de deux entiers valant -1, 0 ou 1. Ces valeurs indiquent de combien les coordonnées de la proie doivent être décalées. Considérant que la première case de la prairie est en haut à gauche, une direction de $(1, -1)$ correspond, par exemple, à un mouvement vers la case en haut $(1)$ et à droite $(-1)$, une direction de $(0, 0)$ correspond à une proie immobile.
Avant chacun de ses déplacements, chaque animal peut opérer un changement de direction avec une probabilité `p_ch_dir`, autrement dit si un nombre aléatoire compris entre 0 et 1 est inférieur à cette valeur. Pour obtenir ce nombre aléatoire, il faut utiliser la fonction `rand` qui renvoie un entier et le diviser par la valeur maximum, à savoir `RAND_MAX`. Rappel : toutes les "probabilités" seront déclarées sous forme de variables globales dans le fichier contenant la fonction main.

1. Ecrivez une fonction permettant de faire bouger tous les animaux contenus dans la liste chaînée passée en argument. Le déplacement de l'animal se fera dans la direction indiquée par le champs `dir`. Le monde sera supposé torique, c'est-à-dire que si la proie essaie d'aller en haut alors qu'elle est sur la première ligne, elle se retrouvera automatiquement sur la dernière ligne. De même si une proie essaie d'aller à droite alors qu'elle est sur la dernière colonne de droite, elle réapparait sur la même ligne et sur la colonne la plus à gauche.

   La fonction aura le prototype suivant :

   ```
   void bouger_animaux(Animal *la);
   ```

   ATTENTION : il faut bien vérifier que les coordonnées sont correctes, et notamment positives. `x=x%10` ne garantit pas que `x` sera positif ! Une erreur à ce niveau peut provoquer des erreurs de segmentation dans la fonction d'affichage, erreurs de segmentation TRES difficiles à détecter (les debogueurs n'y voient que du feu...).

2. Ecrivez une fonction permettant de gérer la reproduction des animaux. Vous parcourrez la liste passée en argument et, pour un animal `ani`, vous ajouterez un nouvel animal à la même position qu'`ani` avec une probabilité `p_reproduce`. Le nouvel animal a pour énergie la moitié de celle de son parent et le parent a son énergie divisée par 2.

Remarque : l'ajout se fera en tête, un animal qui vient de naître ne sera pas considéré par votre boucle et ne pourra donc pas lui-même se reproduire lors de ce pas de temps. Il pourra cependant, bien sûr, se reproduire lors des pas de temps suivants avec la même probabilité que son parent.

La fonction aura le prototype suivant :

```
void reproduce(Animal **liste_animal, float p_reproduce);
```

# TME

## Exercice 3 *(obligatoire)* – Lecture/écriture de fichiers

1. Complétez le main du fichier `main_test2.c` pour écrire l'écosystème créé et le lire dans de nouvelles listes. Vous vérifierez que les écosystèmes lus et écrits sont bien les mêmes. Vous prendrez soin de libérer la mémoire allouée et de vérifier qu'il n'y a pas de fuite mémoire.

## Exercice 4 *(obligatoire)* – Déplacement et reproduction

1. Complétez le main du fichier `main_ecosys.c` pour tester la fonction de déplacement. Vous ne créerez qu'un animal à une position que vous aurez définie et en le déplaçant dans une direction que vous aurez aussi définie. Vérifiez bien la toricité du monde.
2. Complétez le main du fichier `main_ecosys.c` pour tester la fonction de reproduction. Vous mettrez le taux de reproduction à 1 en vérifierez que le nombre d'animaux est bien multiplié par 2 à chaque mise à jour.

## Exercice 5 *(obligatoire)* – Gestion des proies

1. Écrivez une fonction de mise à jour des proies. Cette fonction devra :
   — faire bouger les proies en appelant la fonction `bouger_animaux` ;
   — parcourir la liste de proies :
      — baisser leur énergie de 1,
      — supprimer les proies dont l'énergie est inférieure à 0 ;
   — faire appel à la fonction de reproduction.

   La fonction aura le prototype suivant :

   ```
   void rafraichir_proies(Animal **liste_proie, int monde[SIZE_X][SIZE_Y]);
   ```

2. Dans la fonction main du fichier `main_ecosys.c`, vous créerez 20 proies, puis vous écrirez une boucle qui s'arrête lorsqu'il n'y a plus de proies ou qu'un nombre maximal d'itération est atteint (par exemple 200). Dans cette boucle, vous mettrez à jour les proies et afficherez l'écosystème résultant.

   Pour que vous ayez le temps de voir l'état de votre écosystème, vous pourrez ajouter des pauses en utilisant la fonction `usleep`.

   ```
   man 3 usleep
   ```

   pour avoir une description de son fonctionnement et de la façon de l'utiliser.

## Exercice 6 *(obligatoire)* – Gestion des prédateurs

1. Pour gérer les prédateurs, nous aurons besoin d'une fonction qui va vérifier s'il y a une proie sur une case donnée. Ecrivez cette fonction qui aura le prototype suivant :

   ```
   Animal *animal_en_XY(Animal *l, int x, int y);
   ```

   `l` est la liste chaînée des proies et la valeur renvoyée est un pointeur sur une proie dont les coordonnées sont `x` et `y` (la première rencontrée) ou `NULL` sinon.
2. Les prédateurs sont gérés comme les proies. Comme ils utilisent la même structure, les fonctions `creer_animal`, `ajouter_en_tete_animal`, etc. peuvent être réutilisées telles quelles. Écrivez la fonction de mise à jour de prédateurs qui respectera le prototype suivant :

   ```
   void rafraichir_predateurs(Animal **liste_predateur, Animal **liste_proie);
   ```

   Cette fonction est directement inspirée de la fonction de rafraichissement des proies. Vous pourrez copier-coller celle-ci et faire des modifications. Vous devrez notamment baisser l'énergie d'un prédateur et faire en sorte que, s'il y a une proie située sur la même case qu'un prédateur, elle soit "mangée", permettant ainsi au prédateur de récupérer les points d'énergie de la proie.
3. Modifier votre fonction main (`main_ecosys.c`) pour voir évoluer également les prédateurs.
   Vous pouvez à présent observer l'évolution de votre petit écosystème et notamment tester différentes valeurs des constantes pour étudier l'impact que cela peut avoir sur votre écosystème.

## Exercice 7 *(entraînement)* – Gestion de l'herbe

Nous pouvons enrichir notre écosystème en ajoutant de l'herbe.Les proies mangent de l'herbe, mais la quantité d'herbe est limitée : si une proie mange l'herbe d'une case, il n'y en a plus et l'herbe met un certain temps à repousser. La quantité d'herbe et le temps de repousse sont modélisés par un tableau à 2 dimensions d'entiers. Si la valeur d'une case est positive ou nulle, il y a de l'herbe, sinon non.

1. Dans le fichier `main_ecosys.c` vous déclarerez un tableau statique à 2 dimensions d'entiers de taille `SIZE_X*SIZE_Y` et vous l'initialiserez à 0 dans toutes ses cases.
2. A chaque itération de la simulation, la quantité d'herbe de chaque case est incrémentée de 1. Ecrivez une fonction `void rafraichir_monde(int monde[SIZE_X][SIZE_Y])` qui effectue cette opération.
3. Ajouter la prise en compte de l'herbe dans la fonction de mise à jour des proies. Les proies mangent de l'herbe s'il y en a sur leur case en gagnant autant de points d'énergie qu'il y a d'herbe sur la case. Il faut alors mettre le temps de repousse de l'herbe de la case où est l'animal à `temps_repousse_herbe` (qui est négatif).

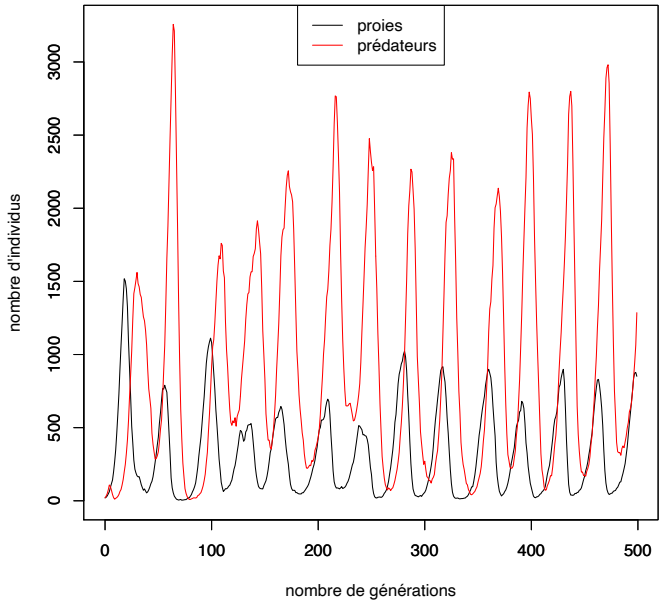
## Exercice 8 *(approfondissement)* – Graphiques de l'évolution des populations

Comme vous l'avez peut-être remarqué, et si vos paramètres le permettent, le nombre d'individus oscille au cours des générations. Nous voulons tracer le graphique du nombre d'individus (proies, et prédateurs si vous les avez simulés) en fonction du nombre d'itérations. Pour cela, il faut écrire dans un fichier à chaque itération une ligne contenant l'indice de l'itération, le nombre de proies, le nombre de prédateurs, séparés par des espaces.

1. Dans le fichier `main_ecosys.c` modifiez votre main pour réaliser cela. Vous pourrez ensuite tracer vos courbes dans un tableur ou avec gnuplot en tapant `gnuplot` puis en tapant

   ```
   plot "Evol_Pop.txt"~using 1:2 with lines title "proies"
   replot "Evol_Pop.txt"~using 1:3 with lines title "predateurs"
   ```

   si vous avez écrit les comptages dans un fichier nommé `Evol_Pop.txt`.
   Vous devriez obtenir une courbe ressemblant à cela :
2. Faites varier les paramètres du modèle et observez les variations de population.


proies
prédateurs
nombre d'individus
0
500
1000
1500
2000
2500
3000
0
100
200
300
400
500
nombre de générations