TD 1 – L'environnement de développement.

Le système avec lequel vous travaillerez est un système UNIX dérivé d'une version de LINUX (Darwin pour être complet). Vous utiliserez le compilateur gcc en version 3.3 et un éditeur de texte que l'on peut appeler de deux façons différentes : soit à l'aide d'un lanceur d'applications, le Dock à droite de l'écran, soit depuis la ligne de commande linux par la commande see (ou smultron suivant le cas) suivie du nom du fichier.

Ce sont ces éléments que nous allons vous présenter lors de ce premier TD, ainsi que la façon d'utiliser le debugger.

Vous serez par la suite amenés à utiliser le logiciel Octave présent sur ces postes.

1/ L'éditeur de texte

Il s'agit d'un programme en licence libre pour les universités et particuliers nommé Smultron ou TextWrangler. Son icône est une fraise/un losange bleu.

Un simple clic sur l'icône dans le Dock (la barre à droite de l'écran) permet le lancement de cet application.

La première action va consister à écrire un programme. Saisissez le code suivant dans la fenêtre :

```
#include<stdio.h>
int main(void)
{ printf("Coucou \n");
  return 0;
}
```

Pour obtenir les caractères "spéciaux" les combinaisons de touches sont les suivantes :

Caractère	Touche	Caractère	Touche
{	alt⌥ + (	]	alt⌥ +⇧+)
}	alt⌥ +)	\	alt⌥ +⇧+ /
[	alt⌥ +⇧+ (	| (pipe)	alt⌥ +⇧+ L

Une fois le code saisi, enregistrez-le soit en déroulant le menu "Fichier" et en choisissant l'item "Enregistrer", soit en tapant au clavier le raccourci "⌘" + S.

Une fenêtre se déroule au dessus du document, naviguez jusqu'au dossier "Bureau".

Enregistrez sous le nom qui vous convient en n'oubliant pas de faire suivre le nom du suffixe ".c" ('c' **minuscule** et non 'C' majuscule).

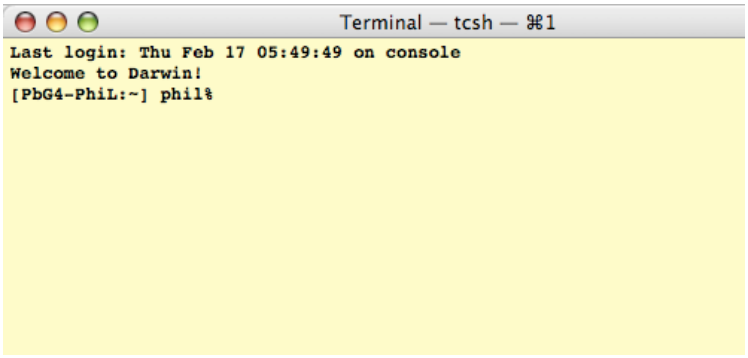
Veillez à ne pas utilisez pas de caractère espace dans le nom du fichier, ni de caractère accentués ou de signe de ponctuation, parenthèse, etc. Vous n'avez le droit qu'aux 26 lettres de l'alphabet, au caractère souligné et aux chiffres.

2/ Le Terminal

Une fois le programme sauvegardé, il vous faudra le compiler pour pouvoir exécuter le programme que vous venez d'écrire. La compilation se fera dans une autre fenêtre appelée "Terminal".

Pour lancer le terminal, cliquez sur son icône dans le Dock (un moniteur noir avec un prompt blanc ">")

Vous devez obtenir une fenêtre du type :



Cette fenêtre est un accès à la ligne de commande UNIX, un peu comme la fenêtre MS_DOS dans un environnement Windows.

La première chose à faire une fois le terminal lancé, est d'aller dans le dossier de travail qui est le Bureau : pour ce faire tapez :

```
cd Desktop
```

Puis vérifiez que vous vous trouvez bien dans le dossier travail : tapez

```
pwd
```

cette commande (Print Working Directory) affiche le chemin d'accès au dossier où vous vous trouvez. Cela doit ressembler peu ou prou à :

```
/Etudiant_1/etudiant/Desktop
```

Notez que contrairement à Windows, le séparateur sous UNIX est "/" et non "\".

Les commandes suivantes peuvent vous être utiles pour travailler sous UNIX :

Commande	Action
pwd	affiche le dossier courant
cd	change de dossier, taper " .." pour revenir au dossier parent
ls	affiche le contenu du dossier courant
clear	efface la fenêtre du terminal (il est possible avec l'ascenseur sur le coté de voir les résultats précédents, par défaut 10 000 lignes sont gardées)
↑	fait défiler les commandes saisies précédemment (remonte dans la liste)
↓	fait défiler les commandes saisies (descend dans la liste)

3/ Le compilateur

Pour compiler un programme il suffit de taper la commande **cc** suivie du nom du fichier.

La version du Shell de l'UNIX que vous utilisez comporte une fonction d'auto-complétion. Cela veut dire que le système est capable s'il n'y a pas ambiguïté de compléter tout seul la commande que vous tapez.

Par exemple, imaginons que vous ayez sauvegardé le programme saisi au point 1/ sous le nom toto.c. Tapez alors

```
cc t
```

puis appuyez sur la touche tabulation ("⇥")

Le nom *toto.c* apparaît alors, vous n'avez plus qu'à valider en appuyant sur la touche "↩".

Le programme est alors compilé et un fichier nommé **a.out** est créé. Ce fichier est le programme. Vérifiez que ce fichier existe bien à l'aide de la commande **ls** (vous devez aussi le voir sur le bureau)

Pour vérifier que le programme fonctionne bien, il suffit de le lancer à l'aide de la commande :

```
./a.out
```

Vous devriez obtenir ceci :

```
Coucou
```

Bien entendu, il est possible que vous ayez commis des erreurs en saisissant le programme. Par exemple, omettez le point virgule après la commande printf dans le programme toto.c. Le code doit être celui-ci.

```
#include<stdio.h>
int main(void)
{ printf("Coucou \n")
  return 0;
}
```

➡ Cliquez dans la fenêtre de Smultron/TextWrangler, et modifiez le code. Puis enregistrez-le.

Notez que si le document n'est pas enregistré, il apparaît un point noir dans le bouton rouge en haut à gauche de la fenêtre de l'éditeur.

➡ Activez le terminal (cliquez sur sa fenêtre ou sur son icône dans le Dock), puis compilez à nouveau le programme.

```
cc toto.c
```

Vous devriez obtenir quelque-chose ressemblant à ceci (suivant la version du compilateur installée) :

```
toto.c: In function 'main':
toto.c:5: error: parse error before 'return'
```

Note Bene:

*Il est possible d'arrêter un programme à tout instant en tapant au clavier "**ctrl + C**". Il est aussi possible de fermer la fenêtre du terminal. Il suffit d'en ouvrir une nouvelle et de ne pas oublier d'aller dans le dossier travail.*

4/ Le debugger

Dans le cas présent l'erreur est simple à trouver. Il se peut que vous soyez confronté à des erreurs complexes à déterminer. Dans ce cas il est particulièrement commode d'utiliser un debugger.

L'utilisation du debugger se fait en deux étapes.

À la compilation il faut ajouter l'**option -g**. Ce qui donne :

```
cc toto.c -g
```

Lancer le debugger pour ce faire il faut taper

```
gdb a.out
```

Le debugger est un programme qui permet l'exécution pas à pas du programme que vous venez d'écrire. Il permet de consulter le contenu des variables pendant que le programme s'exécute pas à pas. Il est possible de poser des points d'arrêts pour que le programme s'exécute jusqu'à un point donné. On peut aussi lister le programme etc.

Le tableau ci-dessous liste quelques commandes du debugger.

Dans ce tableau les instructions peuvent être saisies en entier ou plus simplement en tapant la lettre en gras et entre parenthèses.

(**l**)ist deb,fin	-> liste le programme de la ligne deb à fin, si l'une des bornes est omise sur 10 lignes
(**b**)reak n_ligne	-> crée un point d'arrêt sur la ligne n_ligne
clear n_ligne	-> supprime un point d'arrêt sur la ligne n_line
clear	-> supprime tous les points d'arrêts
(**d**)elete n_break	-> détruit un breakpoint par son numéro
(**n**)ext ou (**s**)tep	-> exécute la ligne suivante, step entre ds les fonctions
(**n**)ext ou (**s**)tep k	-> exécute les k instructions suivantes
(**p**)rint nom_var	-> affiche le contenu de nom_var
display nom_var	-> affiche le contenu de nom_var a chaque arrêt du programme
undisplay nom_var	-> annule display
(**c**)ontinue	-> reprend le cours du programme
(**r**)un	-> exécute le programme jusqu'au premier point d'arrêt
(**q**)uit	-> Quitte le debugger.
info break	-> donne la liste des breakpoints (points d'arrêts)
kill	-> Tue le programme en cours de débogage.

Il y en a d'autres, mais elles sont d'usage plus rare.

Il faut lancer le programme pour pouvoir l'exécuter en pas à pas (n) avec (r)un

Appuyer sur "↩" répète la dernière commande.

Par exemple, compilez le programme *toto.c* en mode *debug*, puis lancez le debugger, et tapez ***l***. Vous devez avoir ceci à l'écran

```
(gdb) l
1       #include<stdio.h>
2
3       int main(void)
4       {       printf("Coucou \n");
5               return 0;
6       }
(gdb)
```

Exécutez pas à pas le programme. Si vous tapez *n* ou *s*, le debugger vous répond que le programme ne tourne pas. Mais si vous tapez *r* pour lancer le programme, comme celui-ci est exempt d'erreurs, il s'exécute jusqu'à la fin.

Pour pouvoir **prendre le contrôle**, nous allons poser un **point d'arrêt (breakpoint)** en tapant :

```
b  4
```

Vous obtenez à l'écran :

```
(gdb) b 4
Breakpoint 1 at 0x1db0: file toto.c, line 4.
```

Si vous lancez le programme en tapant

```
r
```

vous obtenez :

```
(gdb) r
Starting program: /Etudiant_1/Etudiant/Desktop/a.out
Breakpoint 1, main () at toto.c:4
4       {       printf("Coucou \n");
```

Il est alors possible d'exécuter le programme pas à pas en tapant *s* ou *n*.

```
(gdb) s
Coucou
5               return 0;
(gdb)
6       }
(gdb)
Program exited normally.
```

Notez que la commande **s** a été répétée deux fois en appuyant sur entrée **sans re-saisir** la commande. Un simple appui sur la touche [entrée] "↩" suffit, en effet, à répéter la dernière commande.

I : Apprentissages de Base

Exercice 1

Objectif : Ecrire un programme simple qui mette en œuvre les fonctions printf et scanf. utilisation de l'instruction if pour des tests simples, et de switch. Sorties formatée avec printf.

➡ Ecrivez un programme qui demande à l'utilisateur de saisir son âge, puis affiche le message, "vous avez [la valeur saisie] ans".

➡ Ecrivez un programme qui demande à l'utilisateur de saisir deux valeurs entières relatives puis affiche leur somme.

➡ Ecrivez un programme qui demande à l'utilisateur de saisir deux valeurs entières relatives A et B puis affiche leur différence.
Complétez ce programme pour qu'il affiche l'un des trois messages suivant le cas :

```
"La différence A - B est plus grande que A"
"La différence A - B est plus petite que A"
"La différence A - B est nulle".
```

Dans les messages ci-dessus, A et B seront remplacées par les valeurs saisies.

➡ Complétez le programme précédent pour qu'il saisisse des valeurs réelles. L'affichage des valeurs dans les trois messages se fera alors sur 8 caractères et 3 décimales.

➡ Ecrivez un programme qui saisisse deux valeurs puis demande l'opération à effectuer : les réponses possibles seront +, -, * ou /. Vous utiliserez d'abord une série de test puis un switch pour choisir le type de calcul à effectuer.

➡ A ce point vous devez être capable d'utiliser les instructions printf et scanf sans réfléchir à leur syntaxe. Vous devez connaître la syntaxe de l'instruction switch

Exercice 2

Objectif : Ecrire un programme qui permette le calcul d'une somme d'entiers saisis au clavier. découverte des test logiques et des boucles for et while/do while. Règles d'usage de ces différentes boucles.

On se propose d'écrire un programme qui calcule la somme de N entiers entre 0 et 20 La saisie des nombres et le calcul de la somme se feront dans une même boucle.

➡ Dans une première version du programme, N est demandé à l'utilisateur avant la saisie des valeurs

➡ Dans une seconde version du programme, la saisie des valeurs dure tant que l'on n'entre pas un code d'arrêt, ce code sera le nombre 99

➡ Prévoyez un test pour que x, la valeur saisie, reste dans les limites imposées $0 \leq x \leq 20$, d'abord avec le programme du point 1, puis avec le programme du point 2.

Exercice 3

Objectifs : Ecrire un programme qui calcule une moyenne, comprendre l'adaptation de l'opérande à l'opérateur, apprendre à réutiliser un programme précédent.

➡ Editez le programme du point 2 de l'exercice 2. Modifiez-le pour qu'il affiche en fin de programme le nombre de valeurs saisies et la somme.

➡ Calculer la valeur moyenne et affichez-là. Que constatez-vous ?

➡ Quelle solution proposez-vous ?

Exercice 4

Objectifs : Ecrire un programme qui calcule la factorielle d'un **entier** quelconque. Utilisation d'une boucle for décroissante ou d'un test.

On rappelle que la factorielle est une fonction qui à un entier positif N associe :
N! = N*(N-1)*(N-2)*…*2*1, par convention 0! = 1

➡ Ecrivez un programme qui calcule N! d'un entier positif quelconque (0 compris).

➡ Calculez avec votre programme 8! (ou 17! suivant les réglages du compilateur). Que constatez-vous ? Quelle solution proposez-vous pour corriger cette erreur ?

Exercice 5

Objectifs : Programme qui calcule la moyenne de N **entiers**, utilisation de **variables dimensionnées.**

➡ Saisir le nombre de valeurs

➡ Saisir les valeurs dans une boucle.

➡ Calculer la somme des éléments du vecteur dans une autre boucle.

➡ Affichez la valeur moyenne

Exercice 6

Objectif : Utiliser une fonction typée simple.

- ### moyenne :

Reprenez le programme qui calcule la moyenne avec un tableau.

➡ Ecrivez la fonction moyenne avec le prototype suivant :

```
float moy(float somme , int N);
```

Cette fonction calcule la valeur moyenne des éléments d'un vecteur.

➡ Modifiez la fonction main( ) pour qu'elle utilise moy().

➡ Conclusion ?

- ### factorielle :

Reprenez le programme qui calcule la factorielle.

➡ Ecrivez la fonction fact( ) avec le prototype suivant :

```
float fact( int N );
```

Cette fonction calcule la valeur de factorielle N.

➡ Modifiez la fonction main( ) pour qu'elle utilise fact( ).

Utilisation du Debugger (à traiter après la série d'exercices)

Saisissez le programme suivant :

```
#include<stdio.h>

/*
 * Ce programme calcule la somme des chiffres constituant un nombre
 * Par exemple pour 367, le programme doit renvoyer 16.
 *
 * On rappelle que % calcule le reste de la division entiere
 * par exemple 7%2 retourne 1, puisque 7 = 3*2 + 1.
 *
 * Note ce programme comporte des erreurs dans la realisation de l'algorithme
 * Il retourne des resultats errones. A vous de les corriger, Voyez avec
 * le deboggeur, l'evolution des variables :
 * nombre / a / S / n.
 * Corrigez en utilisant ce que vous observez avec le deboggueur
 * le programme.
 * Nous vous conseillons de placer de point d'arret ligne 34...
 *
 */


int main(void)
{  int nombre;
   int a,n,k;
   int S ;

   printf("Entrez le nombre");
   scanf("%d",&nombre);

   n=0;
   while (nombre>10)
   {    a = nombre%10;
        nombre = nombre/10;
        S = S + a;
        n++;
   }

   printf("La somme est : %d \n",S);

   return 0;
}
```

Ce programme comporte des erreurs de syntaxe, et des erreurs d'algorithme.

Vous devez corriger ce programme pour qu'il donne les résultats escomptés (lire les commentaires) en utilisant le debugger, l'éditeur, et le terminal.

II : Mini Projets

Objectifs :

Ce premier mini-projet a pour objectif de vous faire travailler des algorithmes avec plusieurs boucles imbriquées et de vous faire manipuler des fonctions sans paramètres de sortie.

1. Exercice

Objectif : Ecrire un programme qui trace à l'écran un "sapin", le but étant la mise en œuvre d'algorithmes à boucles imbriquées.

- ➡ Ecrivez un programme qui écrive sur N lignes, d'abord une étoile sur la première ligne, puis 2 sur la ligne suivante etc. Autant d'étoiles que le numéro de la ligne en cours. (Fig 1)
- ➡ En modifiant le programme précédent, dessinez un triangle composé d'étoiles. (Fig 2)
- ➡ Ajoutez maintenant un tronc de 3 étoiles sur deux lignes à votre arbre. (Fig 3)

Fig 1	*Fig 2*	*Fig 3*
*	*	*
**	***	***
***	*****	*****
****	*******	*******
*****	*********	*********
		***
		***

2. Mini projet : La bataille navale.

Objectif : Réaliser un programme “bataille navale”. Afin de vous faciliter la tâche une analyse du problème vous est proposée. Respectez les étapes proposées et vous réussirez à réaliser ce programme en 2/3 séances.

2.1. Analyse sommaire du problème :

On réalisera le programme en respectant les consignes suivantes et en trois étapes. Un des problèmes essentiels est le choix du codage des données. Comment représenter le plateau de jeu dans la mémoire de l'ordinateur ?

Ensuite, il faut définir une représentation à l'écran du plateau de jeu. Pour des raisons de temps, vous ferez un quadrillage à base de caractères.

Il faut ensuite que l'ordinateur place les bateaux. Ceci va vous amener à étudier et utiliser le générateur de nombres aléatoires.

Enfin, vous allez gérer le jeu. Saisir les tirs du joueur, gérer le fait qu'il a pu toucher un navire en affichant un message approprié, gérer le fait qu'il a pu couler un navire (= toutes les cases du navire touchées) en affichant un message approprié. Enfin, vérifier que tous les bateaux n'ont pas été touchés auquel cas la partie serait terminée.

2.2. Codage des données :

Les données nécessaires au jeu seront codées de la façon suivante :

Le plateau est une variable dimensionnée à 2 dimensions de type **int**. Le contenu de chaque case peut être le suivant :

Le tableau contient 4 types de valeurs.

0 : On n'a pas testé cette case

x, et x>0 : La case est occupée par le bateau x

x, et x<0 : La case est occupée par le bateau x et il a été touché

999 : la case est vide et elle a été testée.

Note : Pendant la phase de développement du programme, on affiche le plateau comprenant les bateaux avec leurs numéros. Par la suite, on affiche soit **rien** (ce qui signifie que la case n'a pas été testée), soit **O** (= tir dans l'eau), soit **X**(= touché ou coulé).

Vous utiliserez une variable dimensionnée qui contient autant d'éléments que de bateaux +1.

Chaque élément de cette variable contiendra le nombre de cases non touchées du bateau en question. Ainsi, si cette variable porte le nom **bato** et que le bateau 1 n'a pas été touché, **bato [1]** devra contenir la valeur 2 (bateau de 2 cases). Touché une fois, la variable **bato[1]** sera décrémentée[1] (**bato[1] = bato[1]-1**).

Il est rappelé que le contenu des cases est soit un numéro de bateau (positif non touché, négatif touché) soit 0 soit 999.

Note : on ne peut pas utiliser de bateau numéro 0 puisque c'est le code case vide, on a donc obligatoirement le premier bateau qui porte le numéro 1.

3. Affichage de la grille de jeu

La fonction suivante est capable d'effacer l'écran (le terminal) en utilisant des séquences d'échappement (le détail importe peu).

```
void clrscr(void)
{     printf("\x1b[2J \x1b[0;0H");
}
```

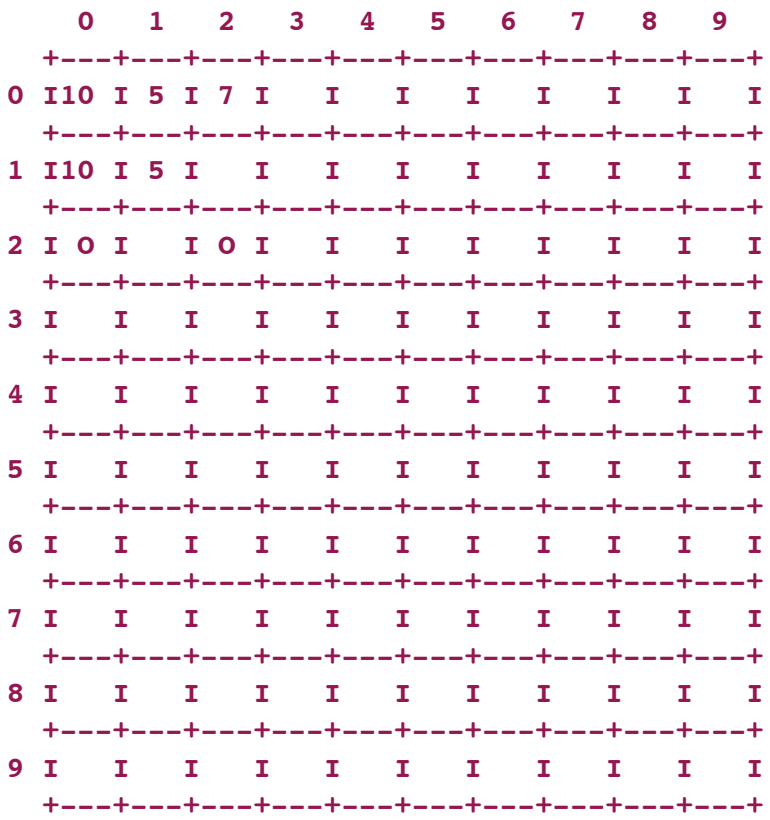
Vous l'inclurez à tous vos programmes. L'appel de **clrscr()** efface l'écran.

Pour le développement du programme, vous afficherez la grille du plateau de jeu avec un exemple de contenu initialisé à la déclaration comme suit :

```
#include <stdio.h>
#define dim 10

typedef int plato[dim][dim];

…

int main(void)
{     plato a={{-10,5,7},{10,5},{999,0,999}};
      …
      return 0;
}
```

[1] incrémenter : augmenter le contenu d'une variable de 1, décrémenter c'est le diminuer de 1.

Ce qui doit conduire à la grille suivante :



La grille est dessinée avec des caractères **+**, **-** et I (i majuscule).

Procédez par étapes !

- Vous commencerez par faire afficher une grille vide. Regardez la grille, elle est une succession de lignes avec des motifs identiques. Utilisez des directives #define pour définir des motifs que vous utiliserez avec profit pour afficher la grille.
- De même la taille de la grille est définie dans une constante en début de programme (dim) cf ci-dessus.
- puis vous prévoirez l'affichage du contenu des cases.
- Enfin, vous ferez de ce code une fonction que vous appellerez **aff**. Réfléchissez-bien aux variables qu'il faudra passer à cette fonction.

Il ne faut pas hésiter à essayer votre code d'affichage pour voir où sont les erreurs, comment passer à la ligne etc.

4. Gestion des tirs sur les bateaux.

4.1. Analyse algorithmique

Pour être capable d'afficher si on a coulé tous les bateaux, on utilisera une variable **NbCases** qui contient le nombre total de cases non touchées et occupées par des bateaux du plateau de jeu. Chaque fois que le joueur touche un des bateaux, on décrémente cette variable. Par conséquent, lorsqu'elle vaut zéro, c'est que tous les bateaux ont été coulés.

En parallèle, une variable “coup” est incrémentée à chaque essai, ce qui permettra de calculer le taux de réussite.

On testera toujours avec l'exemple donné ci-dessus, en positionnant correctement le contenu des différentes variables afin de ne pas perdre de temps (pour l'instant on développe le programme, on jouera plus tard !)

Afin de pouvoir afficher le message "coulé", on crée une variable dimensionnée appelée **bato [ ]** qui pour chaque navire **k** contient le nombre de cases (la longueur) du navire en question. La grille contient, elle, les numéros des navires à la position qui est la leur.

Ainsi, si un tir tombe sur une case contenant la valeur **k**, on décrémente **bato[k]** et on met **-k** dans la case en question. En même temps on diminue **NbCases**.

Dès que **bato[k]** atteint 0, on affiche le message "coulé".

Il ne doit alors plus y avoir de case du plateau de jeu contenant la valeur **k**, mais uniquement des cases contenant la valeur **-k**.

4.2. Réalisation

Le programme saisira le tir de l'utilisateur sous forme de deux coordonnées numériques entières (x, puis y).

Il affichera la nouvelle grille tenant compte de ce tir, le nombre de coups tirés ainsi que le nombre de cases de bateau non touchées restantes.

➡ Ecrivez une fonction gestir

```
int gestir(plato a, int bato[nbr] )
```

qui réalisera les actions suivantes :

- saisie des coordonnées du tir.
- détermination si un navire est touché.

Si oui :

1. mise à jour de la variable bato[ ],
2. affichage touché ou coulé
3. mise à jour du plateau.

Cette fonction **gestir** est typée, elle retourne 0 si aucun navire n'est touché, -1 sinon. Cela afin de mettre à jour la variable **NbCases** dans la fonction **main**. Dans **main** on teste si la variable **NbCases** est nulle, auquel cas la partie est finie.

5. Placement aléatoire des bateaux

5.1. Analyse du problème

Elle consiste en l'étude du générateur de nombres aléatoires et de la génération du tableau de jeu (placer les bateaux)

Objectifs : Étude des fonctions **rand** et **srand**. (nécessitent le header **stdlib.h**). Utilisation de l'opérateur modulo (**%**), choix d'un **germe** aléatoire (**time** et **time_t**)

L'ordinateur est une machine déterministe. Rien n'est aléatoire dans son fonctionnement. Pourtant nous allons lui demander de placer les navires de façon aléatoire. Pour ce faire, nous allons utiliser la fonction **rand**. Cette fonction retourne un nombre de type **int** aléatoire entre 0 et **RAND_MAX**. **RAND_MAX** est une constante définie dans le header **stdlib.h**.

5.2. Exercice :

➡ Ecrivez un programme qui calcule 10 nombres aléatoires à l'aide de la fonction **rand**. Relancez plusieurs fois votre programme. Que constatez-vous ?

Pour éviter ce problème, il existe une fonction qui initialise le générateur de nombres aléatoires. Cette fonction **srand** initialise la série de nombres aléatoires générés par **rand**. Bien entendu, on pourrait saisir au clavier une valeur que l'on choisirait arbitrairement afin d'initialiser le générateur de nombres aléatoires, mais il serait alors facile de tricher…

Pour initialiser ce générateur, on utilisera le seul aléa que l'ordinateur subisse : l'heure de son allumage. Pour ce faire on utilisera la fonction **time** qui est contenue dans le header **time.h** comme suit :

```
time_t t;
// on crée une variable t de type time_t (defini dans time.h)
srand(time(&t));
// on initialise le generateur de nombres aleatoires.
```

➡ Modifiez le programme précédent pour qu'il calcule 10 nombres aléatoires à l'aide de la fonction **rand** initialisée par **srand**. Relancez plusieurs fois votre programme. Que constatez-vous ?

Une fois le générateur initialisé, il faut restreindre les valeurs tirées par le générateur. En effet **RAND_MAX** vaut : 2 147 483 647. Il est hors de question d'avoir un plateau de jeu de plus de deux milliards par deux milliards de cases ! Pour restreindre le plateau à NxN cases, on utilisera la fonction **modulo** (**%**). En effet, quelque soit X, X%N est compris entre 0 et N-1 puisqu'il s'agit du reste de la division entière.

Modifiez le programme précédent pour qu'il calcule 10 nombres aléatoires entre 0 et 10 à l'aide de la fonction **rand** initialisée par **srand** et de l'opérateur **%**.

5.1. Positionnement des bateaux

On peut maintenant placer les navires aléatoirement. Toutefois il faut que le navire reste dans la grille. Tirer un point de départ n'est pas suffisant. L'algorithme suivant permet de placer des navires de façon aléatoire sans déborder du plateau.

Note : Pour simplifier on supposera que tous les bateaux ont la même longueur : 2 cases.

```
Pour le bateau k
      1/ On tire un point de départ, (rand et modulo)
      2/ On tire une direction : un nombre entre 1 et 4 avec la
      convention 1 : le bateau part vers le haut, 2 vers la
      droite, 3 vers le bas, 4 vers la gauche
      3/ Peut le placer dans ce cas ? Il faut : rester dans la
      grille, et que les cases concernées soient vides.
      Si oui on passe au bateau suivant, sinon, retour en 1
fin de pour
```

Attention ! On ne peut placer un bateau qu'à deux conditions

- la case existe bien
- elle est vide

➡ Ecrivez une fonction qui place **nbr** navires dans la variable de type **plato** qui lui est transmise. **nbr** sera définie en constante en début de programme.

6. Quatrième partie

En fait, il n'y a pas de quatrième partie, le programme est fini ! Il n'y a qu'à assembler les morceaux et supprimer l'affichage en mode débogage du plateau de jeu. C'est à dire que la fonction **aff** n'affiche plus le contenu des cases, mais seulement les tirs (à savoir **X** ou **O**)

On peut envisager des améliorations si l'on a du temps afin de vérifier les saisies de l'utilisateur, voire même prévoir des navires de tailles différentes en initialisant les cases de variable **bato** aux dimensions des navires à placer. Il suffit alors dans la fonction écrite au point 5 de vérifier que toutes les cases (on connaît la longueur du navire que l'on place) sont bien libres.

Enfin, on peut imposer le fait que les navires ne doivent pas être contigus. Attention ce n'est pas simple !