|  |
| --- |
| **Note technique du projet Démineur en langage C** |

*Par Timothée BARON*

**Introduction** :

Ce programme a été conçu en langage C avec l'utilisation de procédures et fonctions appelées dans le main ou via certaines procédures elles-mêmes suivant la structure choisie. Sont aussi utilisés, les pointeurs et les tableaux.

Les bibliothèques utilisées sont stdlib, stdio et myconio.

Ci-dessous le schéma simplifiée du code avec les procédures et fonctions appelées :

OU

1. **Main( )**
   * 1. *Variables*
        1. Locales

Tab[ ][ ] : déclaration du tableau à 2 dimensions contenant la grille des valeurs du démineur

choixR : pour stocker la donnée du choix pour recommencer la partie ou non

* + - 1. Générales

Choix de générale car ces variables ont une incidence sur tout le reste de la partie (programme)

m : c’est le nombre de mines déterminées lors de la procédure choix

n : c’est la taille du tableau déterminé lors de la procédure choix

partie : varie selon l’état de la partie (perdu, gagné, recommencer)

* + 1. *Explication Structure*

Le main débute par une boucle *do* en fonction de l’état de la partie par la variable générale *partie*. Ceci permet d’arrêter le jeu à tout moment pour recommencer par la touche *échap* ou en ayant perdu.

Après la détermination de la taille du tableau et du niveau de difficulté et subséquemment du nombre de mines par la fonction choix, les autres procédures principales sont appelées les unes après les autres telles que numérotées dans le schéma (cf schéma du début).

Le srand est initialisé dans le main mais sert uniquement à placer les mines dans la procédure placeMines.

* + 1. *Algorithme synthétique*

*< Boucle tant que partie = 0 (0 signifie continuer)*

*Appel des procédures du jeu (n°1 à 6)*

*Si lors de la partie, partie = 1 (signifie perdu) alors demander si le joueur veut recommencer*

*Idem si partie = 2 ou 3 (échap ou gagné)*

*Si choix de quitter alors partie=0*

*<Fin de la bouche*

1. **Procédure initTab**
   * 1. *Variables*

i : pour désigner l’indice colonne

j : pour désigner l’indice ligne

* + 1. *Paramètre*

Tab[][] : le tableau de base

* + 1. *Explication Structure*

Double boucle imbriquée simple et classique pour les tableaux à deux dimensions.

Choix de valorisation à 10 pour les extrémités du tableau et 0 pour l’intérieur.

* + 1. *Algorithme synthétique*

*< Pour colonne=0, tant que < à taille de tableau initial (carré), incrémente +1 à chaque passage boucle*

*<Idem pour indice ligne*

*Si bordure : colonne,ligne = 10*

*Sinon valorisation à 0*

*<Fin de la boucle ligne*

*< Fin de la boucle colonne*

1. **Procédure affichTab**
   * 1. *Variables*

i : pour désigner l’indice colonne

j : pour désigner l’indice ligne

a, b, c : en type char, pour afficher le caractère spécifié

* + 1. *Paramètre*

Tab[][] : le tableau de base

* + 1. *Explication Structure*

Cette procédure consiste à afficher le terrain de jeu suivant des caractères spécifiques : une croix pour chaque coins, un rectangle rempli, et un espace entre chaque croix ou rectangle. La légende est également affichée du côté droit. L’utilisation de gotoxy est clé dans cette procédure.

La double boucle imbriquée est utilisée de la même façon que pour l’initialisation du tableau mais la limite des boucle est calculée de la façon suivante : 2 x taille – 2 (élimination des extrémités). 2x car à chaque croix ou rectangle il y a un espace.

L’affichage est selon :

* Rectangle si i et j sont pair
* Croix si i et j sont impair
* Espace pour les autres cas
  + 1. *Algorithme synthétique*

*< Pour colonne=0, tant que <* 2 x taille – 2*, incrémente +1 à chaque passage boucle*

*<Idem pour indice ligne*

*Affichage caractère selon valeur de i et j*

*<Fin de la boucle ligne*

*Saut de ligne*

*< Fin de la boucle colonne*

*Affichage de la légende*

1. **Procédure choix**
   * 1. *Variable*

t : pour stocker le choix entré par l’utilisateur

* + 1. *Paramètre*

Aucun

* + 1. *Explication Structure*

Une boucle pour vérification de chaque choix : une pour la taille du tableau et l’autre pour le niveau de difficulté.

Le niveau de difficulté choisi va déterminer le nombre de mines en fonction de la taille selon le calcul suivant : (taille corrigée des extrémités fictives) au carré x 10%.

Chaque choix va entrainer un remplacement de la valeur de n puis de m via une structure de switch.

* + 1. *Algorithme synthétique*

*< 1ère Boucle faire pour test : taille*

*affichage : demande taille*

*entrée de la taille choisie*

*switch selon le choix et stockage de n*

*<Fin de la boucle test*

*<2ème Boucle faire pour test : difficulté*

*affichage : demande difficulté*

*entrée de la difficulté choisie*

*switch selon le choix choix et stockage de m*

*<Fin de la boucle test*

1. **Procédure affichSolu**
   * 1. *Variables*

i : pour désigner l’indice colonne

j : pour désigner l’indice ligne

a, c : en type char, pour afficher le caractère spécifié

* + 1. *Paramètre*

Tab[][] : le tableau de base

* + 1. *Explication Structure*

Cette procédure est appelée quand le joueur a perdu. Elle affiche la solution après le tableau initial (à l’emplacement de la légende maintenant désuète), après avoir « nettoyé » l’espace par des espaces. L’affichage de la solution est de même style que affichTab, sauf les rectangles qui ne sont pas affichés. En effet, cet affichage consiste à afficher l’emplacement des mines seulement.

Suivant l’emplacement du couple (i,j), i et j de tab[ ][ ] sont convertis pour obtenir la bonne valeur emplacement graphique/emplacement de tab[ ][ ].

* + 1. *Algorithme synthétique*

*< Pour colonne=0, tant que <* 2 x (taille – 1)*, incrémente +1 à chaque passage boucle*

*<Idem pour indice ligne*

*Affichage caractère espace après le tableau initial pour nettoyage avant affichage solution*

*<Fin de la boucle ligne*

*Saut de ligne*

*< Fin de la boucle colonne*

*< Pour colonne=0, tant que <* 2 x (taille – 1)*, incrémente +1 à chaque passage boucle*

*<Idem pour indice ligne*

*Replacement curseur suivant i et j*

*Affichage caractère selon valeur de i et j de tab[i/2][j/2]*

*<Fin de la boucle ligne*

*Saut de ligne*

*< Fin de la boucle colonne*

1. **Procédure placeMines**
   * 1. *Variables*

i : pour désigner l’indice colonne

j : pour désigner l’indice ligne

a : compteur de boucle

mines : pour compter les mines placées

* + 1. *Paramètre*

Tab[][] : le tableau de base

* + 1. *Explication Structure*

Cette procédure ce décompose en 3 phases :

* Placement des mines avec i et j générés aléatoirement suivant un intervalle (n-2)+1 (ex : de 1 à 10 si taille est 12 x 12).
* Phase de contrôle du nombre de mines et ajout si mine manquante
* Phase de comptage de mines périphériques par appel de comptPerif
  + 1. *Algorithme synthétique*

*< Pour a=0, tant que a<* m, *incrémente +1 à chaque passage boucle*

*Génération aléatoire de i,j et placement mine à tab[i][j]*

*< Fin de la placement mines*

*< Boucle tant que mines placées < m*

*< Pour colonne=0, tant que < à taille de tableau initial, incrémente +1*

*<Idem pour indice ligne*

*Si tab[i][j] contien une mines alors mine++*

*<Fin de la boucle ligne*

*< Fin de la boucle colonne*

*<Si mine<m*

*Alors placement d’une mine avec coordonnées aléatoires*

*<Fin si*

*< Fin de la boucle tant que*

*< Pour colonne=0, tant que < à taille de tableau initial, incrémente +1*

*<Idem pour indice ligne*

*Si tab[i][j] = 0, alors appel comptPerif et inscription de la valeur suivant les coordonées*

*<Fin de la boucle ligne*

*< Fin de la boucle colonne*

1. **Fonction comptPerif**
2. *Variables*

i : pour désigner l’indice colonne

j : pour désigner l’indice ligne

comptM : pour compter le nombre de mines autour

1. *Paramètres*

Tab[][] : le tableau de base

colM : récupère l’indice colonne

ligneM : récupère l’indice ligne

1. *Explication Structure*

Cette fonction consiste à compter le nombre de mines périphériques et est appelée par placeMines. La fonction renvoi le nombre de mines périphériques.

1. *Algorithme synthétique*

*< Pour colonne=0, tant que < à taille de tableau initial, incrémente +1*

*<Idem pour indice ligne*

*Si tab[i][j] contien une mines alors mine++*

*<Fin de la boucle ligne*

*< Fin de la boucle colonne*

1. **Procédure degagC**
2. *Variables*

i : pour désigner l’indice colonne

j : pour désigner l’indice ligne

1. *Paramètres*

Tab[][] : le tableau de base

colD : récupère l’indice colonne

ligneD : récupère l’indice ligne

1. *Explication Structure*

Cette procédure est récursive. En effet, elle s’appelle elle-même quand elle trouve la valeur 0 sur les contours. A chaque valeur 0 trouvée, elle est remplacée par 11 (signifie dégagée) et la procédure est à nouveau appelée via ses coordonnées. La récursivité s’arrête quand il n’y a plus de 0 mais que des 11.

1. *Algorithme synthétique*

*< Pour colonne=colD-1, tant que < colD+1à taille de tableau initial, incrémente +1*

*<Idem pour indice ligne*

*Si tab[i][j] entre 0 et 9 exclus alors affichage de la valeur*

*Si tab[i][j] = 0 ou > 11 alors tab[i][j]=11 et espace et appel degagC avec nouvelles coordonnées*

*<Fin de la boucle ligne*

*< Fin de la boucle colonne*

1. **Procédure fouille**
2. *Variables*

aucune

1. *Paramètres*

Tab[][] : le tableau de base

colF : récupère l’indice colonne

ligneF : récupère l’indice ligne

1. *Explication Structure*

Suivant la case sélectionnée pour la fouiller, la valeur de la case va entrainer une action différente :

* Si = 0 alors dégagement des contours par appel de la procédure degagC
* Si entre 0 et 9 exclus, alors affichage de la valeur
* Si = 9 (mine) alors partie = 1 (perdu)

1. *Algorithme synthétique*

*Cf explication*

1. **Procédure target**
2. *Variables*

i : pour désigner l’indice colonne

j : pour désigner l’indice ligne

cptM : pour compter le nombre de mines correctement marquées

cM : comptage de mines peu importe si correct ou non (pour affichage)

1. *Paramètres*

Tab[][] : le tableau de base

colT : récupère l’indice colonne

ligneT : récupère l’indice ligne

1. *Explication Structure*

Cette procédure consiste à gérer le marquage des mines sur la grille graphique (touche e) suivant le tableau suivant :

|  |  |  |
| --- | --- | --- |
|  | Marquée | Non marquée |
| Mine | Valeur à 9 (réinitialisation) | Valeur à 99 |
| Pas de mine | Valeur à -20 (idem) | Valeur à +20 |

A chaque marquage le nombre de mines marquées est contrôlé, jusqu’à la victoire et un autre contrôle a lieu pour compter tous les marquages (bons ou mauvais) pour l’affichage.

1. *Algorithme synthétique*

*If suivant les cas du tableau avec des else à chaque passage de cas différent*

*Boucle for pour compter le nombre de mines réelle > jusqu’à victoire*

*Boucle for pour compter les mines marquées (bonnes ou mauvaises)*

*Affichage des mines restant à trouver suivant le marquage*

1. **Procédure moove**
2. *Variables*

x : pour désigner le coordonné de gotoxy en colonne

y : pour désigner le coordonné de gotoxy en ligne

touche : en char pour stocker la touche appuyée via getch

1. *Paramètre*

Tab[][] : le tableau de base

1. *Explication Structure*

Cette procédure est basé sur le getch qui va détecter quelle est touche est appuyée (touches directionnelles, marquage, fouille et échap pour quitter). Pour chaque touche qui entre en jeux, une structure de switch avec les case des touches appuyées.

Pour l’appui sur les touches directionnelles, la condition est que si le curseur est positionné sur une bordure alors le mouvement est bloqué mais il subsiste un défaut si on appui plusieurs fois sur gauche alors que l’on est sur l’extrémité gauche.

1. *Algorithme synthétique*

*<boucle tant que partie = 0 (continuer)*

*Switch*

*Case touches directionnelles > mouvement selon la touche appuyée et selon bordure ou pas*

*Case touche fouiller > Appel de la procédure fouille*

*Case touche marquer > Appel de la procédure target*

*Case touche échap > On quitte*

*<fin boucle*