INF1005C - PROGRAMMATION PROCÉDURALE Travail dirigé No. 4

Fonctions, Tableaux

Objectifs : Permettre à l'étudiant de saisir le concept de fonctions, transmission de paramètres, et les tableaux.
Durée : Deux séances de laboratoire.
Remise du travail : Jeudi 30 octobre à 23h30
Travail préparatoire : Lecture des exercices et rédaction des algorithmes.
Documents à remettre : Sur le site Moodle des travaux pratiques, vous remettrez l'ensemble des fichiers .cpp compressés dans un fichier .zip en suivant la procédure de remise des TDs.
Retard : Les retards ne sont pas tolérés, un retard méritera la note 0.

Directives particulières :

- Une fonction ne devrait pas lire du clavier ou afficher à l'écran, directement ou indirectement, sauf si spécifié dans sa description.

- Vous pouvez ajouter des fonctions, par rapport à celles décrites dans l'énoncé, pour améliorer la lisibilité du code ou pour suivre le principe « don't repeat yourself ».

- Inutile de chercher à afficher les caractères accentués.

- Vous pouvez déclarer toutes les variables désirées.

- N'oubliez pas de mettre les entêtes de fichiers et de fonctions, de respecter le guide de codage (voir la dernière page pour la liste des points) et de placer des commentaires dans le code aux endroits appropriés.

- Utilisez le type **string** pour les chaînes de caractères.

- Il est interdit d'ajouter une attente à la fin de l'exécution du programme; pour exécuter votre programme, utiliser l'option « exécuter sans débogage » dans l'environnement de développement (ou ctrl-F5), pour que l'environnement fasse lui-même l'attente à la fin, ou utiliser un point d'arrêt.

Exercice 1 : Game of sticks

Dans ce jeu, il y a un tas de petits bâtons sur une table. Chaque joueur peut prendre entre 1 et 3 bâtons par tour. Celui qui ramasse le dernier bâton a perdu.

Voici un exemple avec un joueur contre l'ordinateur (en sous-ligné sont les entrées au clavier) :

```
Bienvenu dans le 'game of sticks' !
Quel est le nombre de batons initial (10-100) ? 10
Voulez-vous jouer contre l'ordinateur (1) ou a deux (2)  (1-2) ? 1
Quel est votre nom ? Bozo
Il y a 10 batons sur la table.
Bozo : combien de batons voulez-vous prendre (1-3) ? 1
Il reste 9 batons sur la table.
L'ordinateur a pris 2 batons.
Il reste 7 batons sur la table.
Bozo : combien de batons voulez-vous prendre (1-3) ? 2
Il reste 5 batons sur la table.
L'ordinateur a pris 3 batons.
Il reste 2 batons sur la table.
Bozo : combien de batons voulez-vous prendre (1-2) ? 1
Il reste 1 batons sur la table.
L'ordinateur a perdu.
```

Le programme à écrire dans cet exercice doit implémenter deux versions de ce jeu. Dans la première vous jouerez contre l'ordinateur. Dans la deuxième il y aura deux joueurs que vous créerez et qui joueront l'un contre l'autre.

Au commencement du jeu, vous afficherez un message de bienvenue et vous demanderez les informations sur la partie, soit le nombre de bâtons initial (entre 10 et 100) et si l'utilisateur souhaite jouer contre l'ordinateur ou un autre joueur (1 ou 2 joueurs).

Chaque entrée de nombre (c'est-à-dire toutes les entrées autres que les noms des joueurs) doit être vérifiée. Si l'entrée n'est pas un nombre ou n'est pas dans l'intervalle, vous devez demander à l'utilisateur d'entrer de nouveau l'information et ce jusqu'à temps que l'information soit valide.

Cet exercice est sous la même forme que sera le contrôle pratique.

Le but de cet exercice est d'écrire des fonctions dans un cadre déjà fixé (on connaît déjà les fonctions à écrire ainsi que leurs prototypes). Pour vous aider à vérifier que chaque fonction est bonne, sans avoir terminé d'écrire le programme entier, nous avons conçu un système vous permettant de tester vos fonctions. Lisez attentivement la méthodologie proposée pour réaliser cet exercice.

La forme de la solution VisualStudio (fichier .sln) et des projets qu'elle contient est :

- Le projet « jouer » est un squelette du programme à faire. Il faut compléter ce programme aux endroits indiqués par des commentaires « TODO » dans le programme. Il contient plusieurs fichiers « .cpp », pour séparer le programme en différentes parties. Chaque fichier « .cpp » contient une fonction « principale » pour cette partie du programme et une fonction « test_... »

qui tente de vérifier si la fonction principale fait ce qu'il faut. Il contient aussi un fichier « .hpp » avec les prototypes des fonctions que vous devez implémenter, qui n'a pas à être modifié (ne modifiez pas les prototypes déjà présents).

- Le projet « jouer_qui_compile » contient une version complète du programme que vous devez réaliser, et vous pouvez l'exécuter, mais sans pouvoir visualiser les instructions de ce programme. Ce projet vous permet d'exécuter les fonctions que vous avez écrites, et utiliser les fonctions du solutionnaire pour celles que vous n'avez pas encore écrites. Pour chaque fichier « .cpp » qui compile, le projet utilisera votre fichier, et pour ceux qui ne compilent pas, il utilisera la version du solutionnaire. Des messages classifiés dans les « avertissements » (icône jaune) dans la « Liste d'erreurs » seront affichés pour vos fichiers qui n'ont pas pu être compilés.

- Vous n'avez pas à comprendre le projet « Z test ». Il contient les fonctions de base qui permettent d'écrire plus facilement les fonctions de test et afficher les résultats en couleur.

Au départ, chaque fichier « .cpp » contient une ligne « #error Commenter cette ligne pour compiler ce fichier. », qui force une erreur de compilation, et par le fait même le projet « jouer_qui_compile » à utiliser le solutionnaire (fichiers de fonctions compilées). Lorsque vous commencez à implémenter vos fonctions, il faut évidemment commenter/retirer cette ligne des fichiers correspondants. Si une fonction que vous avez écrite, fait planter l'exécution, et que vous voulez tout de même tester les autres fonctions que vous avez écrites, il peut être utile de remettre ce « #error » dans la fonction problématique.

Méthodologie proposée pour cet exercice :

Commencer par exécuter le solutionnaire : choisir « jouer_qui_compile » comme projet de démarrage et exécuter (avec F5 ou Ctrl F5). Vous pouvez alors visualiser l'exécution du programme complet, vous aidant à mieux comprendre le résultat demandé. Pour chaque fonction, l'affichage « Test … : version du solutionnaire », en jaune, indique que les fonctions du solutionnaire sont utilisées, et non les vôtres.

Choisir une fonction que vous voulez écrire, et ouvrir le fichier correspondant.

Afficher la « Liste des tâches » pour voir ce qu'il y a à faire (la liste affiche uniquement les tâches du fichier actuellement ouvert).

Commenter la ligne « #error … ».

Afficher la « Liste d'erreurs » pendant que vous écrivez la fonction, pour vérifier si vous avez des erreurs qu'« Intellisense » peut détecter.

Pour compiler plus rapidement le projet, compiler uniquement le fichier dans lequel vous travaillez, en utilisant Ctrl F7. Puis vérifier la « Liste d'erreurs ».

Lorsqu'il n'y a plus d'erreur dans ce fichier, exécuter (avec F5 ou Ctrl F5); noter que c'est encore le projet « jouer_qui_compile » qui est choisi comme projet de démarrage.

Pendant l'exécution, vérifier l'affichage pour le test de la fonction que vous avez écrite. Si c'est écrit « version du solutionnaire », terminer l'exécution du programme et aller dans la « Liste d'erreurs » pour voir pourquoi le projet n'a pas utilisé votre version du fichier. Si la compilation du fichier avec Ctrl F7 n'avait donnée aucune erreur, et que la version du solutionnaire est utilisée, c'est généralement que vous avez modifié les types des paramètres ou le nom de la fonction. Noter qu'il y aura plusieurs

avertissements « Fichier non compilé ou ne compile pas. » (un par fichier que vous n'avez pas encore écrit), regardez uniquement ce qui concerne le fichier que vous avez écrit.

Si c'est votre version qui s'exécute, vous verrez l'exécution des tests. Chaque test indique le numéro de ligne où le test est fait (dans la fonction « test_... »), quelle est la comparaison faite en indiquant les noms des variables, quelle est la comparaison faite en indiquant les valeurs, et est-ce que c'est bon (oui) ou pas (non). Par exemple :

```
35: resultat == 2 (1 == 2) ? non
```

Indique qu'à la ligne 35 du fichier qui correspond à la fonction il faut que « resultat == 2 ». Ce programme a une erreur car « resultat » est « 1 », et la comparaison « 1 == 2 » est fausse, d'où le « non » en rouge. Il faut regarder les lignes précédentes dans le programme pour savoir comment la fonction a été appelée pour obtenir ce mauvais résultat. Si les lignes précédentes sont par exemple :

```
simulerCinCout("1\n8\n2\n4\n");
resultat = demanderNombreDansIntervalle(2, 7);
```

Ici, le test qui est fait est de demander un nombre entre 2 et 7 (une fonction que vous devez écrire), en simulant que la personne entre au clavier 1, 8, 2, 4, en appuyant sur « enter » entre chaque valeur. La fonction « demanderNombreDansIntervalle » devrait rejeter les deux premiers nombres et accepter le troisième comme étant valide avant même que la personne ait le temps d'entrer la quatrième valeur. Le résultat devrait donc être 2.

Pour la fonction « jouer », où c'est l'affichage qui est important, le projet affiche à l'exécution « 'cout' correct. » ou « N'a pas trouve '…' dans '…' »; une recherche textuelle est faite dans ce qui est envoyé à « cout », votre programme doit donc utiliser les mêmes mots que la solution. Il est bon de vérifier le programme de la fonction « test_jouer » pour identifier les textes qui sont cherchés. Noter que ces tests textuels ne cherchent que des valeurs et mots clés dans le texte affiché par la fonction, et ne vérifie pas le sens du texte. On peut aussi essayer à la main le programme résultant pour vérifier le bon déroulement de celui-ci.

Après la série de tests sur la fonction, l'exécution du projet affiche « …/… ont réussi » pour indiquer le nombre de ces tests réussis.

Pour effectuer plusieurs de ces tests, la fonction de test prend le contrôle de « cin » et « cout » pour envoyer des entrées automatiquement aux fonctions et vérifier le résultat (précisément, le contrôle est pris entre le « simulerCinCout » et la vérification du résultat par « ESPERE_EGALITE » ou « verifieCout »). Si le programme n'affiche pas les résultats escomptés, soit à cause d'une boucle infinie, ou qu'il plante durant la fonction « test_... » pour la fonction que vous écrivez, vous pouvez décommenter la ligne pour faire des tests à la main dans cette fonction « test_... ». Vous pouvez aussi y ajouter vos propres instructions pour faire des tests. Si vous voulez afficher à l'écran des messages d'information sur l'exécution de votre programme pendant les tests, vous pouvez afficher sur « cerr » plutôt que « cout » (« cerr » fonctionne exactement comme « cout », mais n'est pas capturé par la fonction de test).

Les fonctions à écrire sont :

`int demanderNombreDansIntervalle(int minimum, int maximum)`

Demande un nombre dans l'intervalle, en affichant uniquement « (*minimum-maximum*) ? », pour laisser l'affichage de la question à la fonction qui veut le nombre. Cette fonction doit faire les vérifications et traiter les erreurs en entrée et redemander le nombre s'il est incorrect. Elle a comme valeur de retour le nombre entré.

`int demanderNombreStick()`

Demande le nombre de bâtons, entre 10 et 100, en affichant un message et appelant la fonction ci-dessus. Elle a comme valeur de retour le nombre de bâtons entré.

`int demanderNomsJoueurs(string nomDesJoueurs[2])`

Demande le nombre de joueurs (un nombre entre 1 et 2), encore une fois en appelant la bonne fonction. Elle demande ensuite le nom du ou des joueurs, et place ces noms dans le tableau « nomDesJoueurs »; s'il n'y a qu'un joueur, le deuxième sera mis à « L'ordinateur » (doit être exactement ce nom, sans espace). Elle a comme valeur de retour le nombre de joueurs.

`void jouerHumain(const string& nom, int& nSticks)`

Demande au joueur 'nom' combien de bâtons il veut prendre (ce nombre doit être valide, soit entre un et trois mais pas plus que le nombre actuel de bâtons sur la table 'nSticks'), puis réduit 'nSticks' du nombre de bâtons pris par le joueur.

`void jouerOrdinateur(int& nSticks)`

L'intelligence artificielle de l'ordinateur est très simple : s'il peut gagner immédiatement, il le fera, sinon il va prendre un nombre aléatoire de bâtons. On peut gagner immédiatement si en prenant entre 1 et 3 bâtons on peut s'assurer qu'il n'en reste qu'un seul (que l'adversaire sera obligé de prendre et perdre).

`void jouer(int nSticksInitial, int nJoueurs, const string nomDesJoueurs[2])`

Affiche le nombre initial de bâtons puis fait jouer en alternance les joueurs ou le joueur et l'ordinateur (selon le nombre de joueurs 'nJoueurs'). Dès qu'il reste 1 ou 0 bâtons, la partie termine et est affiché qui « a pedu ». S'il reste 1 bâton, c'est le prochain à jouer qui a perdu puisqu'il doit nécessairement prendre ce bâton, et s'il ne reste aucun bâton c'est que le dernier joueur a fait exprès pour perdre (puisqu'il restait plus d'un bâton, il pouvait ne pas prendre le dernier).

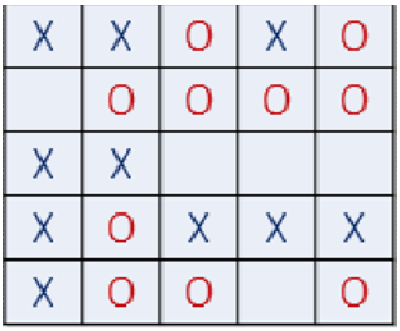
Fonction à ajouter pour la lisibilité :

Dans 'jouerOrdinateur', vous avez probablement une expression un peu compliquée à lire pour générer le nombre aléatoire dans le bon intervalle. Donc vous pouvez ajouter une fonction pour générer un nombre entier aléatoire dans un intervalle quelconque, avec un nom de fonction qui permet de comprendre ce qui est fait. Ajoutez cette fonction immédiatement au-dessus de 'jouerOrdinateur' (pour ne pas avoir à ajouter de prototype à cette fonction), et remplacez l'expression dans 'jouerOrdinateur' par un appel à cette fonction.

Exercice 2 : Modélisation de la ségrégation

Cet exercice est un peu plus libre que le premier pour vous permettre d'apprendre à choisir les types des paramètres de fonctions (il n'est donc pas possible de vous fournir des fichiers avec les solutions, puisqu'on ne peut pas deviner exactement l'ordre, les types et les modes de transmission des paramètres des fonctions que vous allez écrire). Dans cet exercice vous allez apprendre à modéliser le concept de ségrégation, tout en ayant la possibilité d'ajuster quelques paramètres pour étudier leur influence sur le modèle.

Nous allons supposer qu'il y a des « agents » de deux types. Ces types peuvent correspondre à une origine ethnique, un statut économique, etc. Initialement, on place ces agents au hasard sur une grille, chaque cellule de la grille étant soit occupée par un agent, soit vide.



Maintenant il s'agit de savoir si les agents sont satisfaits ou non de leur emplacement. On dira qu'un agent est satisfait s'il est entouré d'au moins *t* pour cent d'agents du même type que lui (en regardant les cases occupées par des agents parmi les 8 cases adjacentes à celle de l'agent, incluant les diagonales). Par exemple, la croix tout en haut à gauche a seulement 2 voisins dans des cases adjacentes occupées. Dans l'exemple, cet agent croix est entouré de 50% d'agents du même type : la croix à sa droite est de même type, et le rond en diagonale n'est pas de même type, les cellules vides ne comptent pas, donc ça donne 1 voisin de même type sur 2 voisins, soit 50%. Si *t*, qu'on appelle le seuil, est fixé à 70%, cela signifie que l'agent (représenté ici par la croix) n'est pas satisfait de son emplacement, car il n'a pas assez d'« amis » (agents de même type) par rapport aux « étrangers » (agents d'un autre type). En espérant qu'il soit plus satisfait, on va le déplacer, et pour ce faire on choisit aléatoirement l'une des cases non encore occupée pour y mettre l'agent, l'ancienne case devenant inoccupée. On continue ainsi jusqu'à ce que tous les agents soient satisfaits.

Suivez les « TODO » dans le fichier « td4_2.cpp ». Les fonctions ont été placées dans l'ordre où on devrait les programmer pour pouvoir les tester facilement à mesure. Commencer par lire le prochain « TODO » de la fonction « test_tout », et s'il requiert de programmer une fonction, suivre les « TODO » de cette fonction, ajouter son prototype au bon endroit et terminer d'écrire ce qu'il faut pour ce « TODO » dans « test_tout ». Compiler et exécuter pour vérifier si cela fonctionne correctement avant de passer au prochain « TODO ». Effacer le « TODO: » au début du commentaire lorsque c'est fait (pour vous rappeler que c'est fait et qu'il ne soit plus dans la « Liste des tâches »), mais garder le reste du commentaire pour vous rappeler de ce que cette partie de programme devait faire (normalement le commentaire serait enlevé au complet, mais dans un premier cours de programmation ce sera plus facile pour vous de suivre ce qui a été fait en gardant le reste du commentaire).

Points du guide de codage à respecter :

Tous les points du guide nommés au TD3 devraient normalement encore être respectés, mais voici un résumé des points sur lesquels nous insistons pour ce TD :

3 et 5 : noms des variables et des fonctions en lowerCamelCase
33 : ajouter un entête de fichier
47 : pas plus d'une signification par variable
48 : aucune variable globale (les constantes globales sont tout à fait permises)
50 : mettre le & près du type
52 : variables vivantes le moins longtemps possible
62 : pas de nombres magiques dans le code
85 : mieux écrire plutôt que commenter
89 : entêtes de fonctions; y indiquer clairement les paramètres [out] et [in,out]

Ainsi qu'un point qui n'est pas actuellement dans le guide : « don't repeat yourself ». Ceci signifie qu'il faut tenter d'éviter d'écrire la même chose plus d'une fois dans un programme. Les instructions de répétition ainsi que la définition de fonctions sont des outils que vous avez vus et qui permettent de réduire les répétitions d'écriture dans le programme.