POLYTECH PARIS-SUD

Et3 - Tronc commun

Programmation C

TP3 - Fonctions

1 Objectifs :

— Consolider sa maîtrise des fonctions
— Savoir décomposer un problème en fonctions adéquates
— Savoir repérer les zones à risque et mettre en place des tests
— Savoir gérer les erreurs d'utilisation
— Savoir aller chercher une information sur internet

2 Rappels : Création et appel d'une fonction

1. Écrivez une fonction qui permet d'afficher « Hello World ! » dans le terminal lorsqu'on l'appelle. Veillez tout particulièrement à respecter les conventions vues lors des premiers TPs ! Aides :
   — Vous devriez avoir 3 fichiers distincts à la fin de cet exercice.
   — Reprenez ce que vous avez fait dans les TPs précédents : cela peut s'avérer bien utile.
2. À présent, améliorez cette fonction pour qu'elle affiche une chaîne de caractères quelconque entrée via le terminal (cf. fonction `scanf` vue dans les précédents TPs).

3 Arguments, retour et gestion d'erreur : le code de César

En réalité, un caractère (variable de type `char`) permet de stocker des nombres. Ces nombres sont compris entre 0 et 255 lorsque le type est `unsigned char` et entre -128 et +127 lorsque le type est `signed char`). Si vous utilisez le type char sans spécifier `signed` ou `unsigned`, ceci sera décidé par le compilateur (gcc). La mémoire ne peut en effet stocker que des nombres. Il y a donc une table qui permet de convertir ces nombres en lettres (voir table ASCII). Le langage C permet de faire cette conversion facilement. Le caractère 'a' est par exemple codé par le nombre 97, 'b' par 98, etc. Pour vous en convaincre, vous pouvez essayer le code suivant :

```
char unChar = 'a';
int unInt = 98;
printf("unChar = %c ou %i", unChar, unChar);
printf("unInt = %c ou %i", unInt, unInt);
```

1. Écrivez une fonction qui prend comme argument un caractère et un entier. Cette fonction devra remplacer ce caractère par la lettre suivante dans l'alphabet située à la distance donnée par l'entier en argument et rendre le résultat (ex : 'a' + 3 = 'd').
   Aides :
   — N'utilisez que des lettres minuscules
   — Attention à bien reboucler sur le 'a' après le 'z'
   — Étant tous les 2 des types d'entier, plusieurs opérations fonctionnent entre un `char` et un `int` (pas besoin de « caster »). En particulier l'addition et la soustraction (ex : `printf("%c/%i", 'a'+1, 'a'+1); //Affiche "b/98"`).

2. Écrivez une seconde fonction qui prend comme argument un entier et une chaîne de caractères (`char*`) ne contenant que des minuscules sans espaces. Cette fonction devra décaler tous les caractères de la chaîne d'autant de lettres dans l'alphabet que l'entier en argument et retourner la nouvelle chaîne obtenue.
   Aides :
   — Avec la librairie `string.h`, vous avez accès à la fonction `strlen(char*)` qui vous permet de connaître la taille de votre chaîne de caractères.
   — On accède à une lettre d'une chaîne de caractères avec `[indice]`. Exemple : `char* mot="HelloWorld"; //alors mot[0]='H', mot[1]='e', etc.`
   — Attention à l'allocation statique (cf. `malloc()`). Si vous n'avez pas encore vu ce que c'est, vous pouvez résoudre cet exercice en modifiant directement votre variable (`char*`) en paramètre.
   — Utiliser la fonction précédente pourrait être un choix judicieux !
3. Écrivez une troisième fonction qui permettra à l'utilisateur de rentrer dans le terminal des mots à coder (autant qu'il le souhaite), des clés de chiffrement associées et affichera les résultats dans le terminal.
   Aides :
   — On prendra bien soin de prévoir de sortir du programme avec une condition d'arrêt.
   — Rappelez-vous que pour initialiser une chaîne de caractères, on peut lui donner une taille importante. Ex : `char mot[100] ;`
4. D'après vous, quel(s) intérêt(s) y a-t-il à effectuer ce genre de découpage en plusieurs fonctions ?

Gestion d'erreur : nous définirons ici la gestion des erreurs comme étant un moyen pour empêcher les utilisateurs de faire n'importe quoi avec votre programme, et bien entendu de les informer un minimum sur ce qu'ils doivent faire. Il y a des similitudes évidentes entre gestion des erreurs et tests (que nous verrons plus loin), mais à la différence des tests, la gestion des erreurs doit bien prendre en compte qu'elle s'adresse à des utilisateurs et non à des développeurs.

5. Modifiez votre/vos fonction(s) pour que votre programme empêche les utilisateurs de rentrer des mots avec autre chose que des lettres minuscules sans espace et qu'il leur retourne également des messages informatifs selon le type d'erreur. Cela ne doit pas stopper votre programme pour autant.

4 Tests unitaires : La légende de l'échiquier de Sissa (3000 ans av. JC)

Le roi Belkib promit une récompense fabuleuse à qui lui proposerait une distraction qui le satisferait. Lorsque le sage Sissa lui présenta le jeu d'échecs, le souverain, demanda à Sissa ce que celui-ci souhaitait en échange de ce cadeau extraordinaire. Sissa demanda au prince de déposer 1 grain de riz sur la première case, 2 sur la deuxième, 4 sur la troisième, et ainsi de suite pour remplir l'échiquier en doublant la quantité de grain à chaque case (N.B. 64 cases pour l'échiquier). Le prince accorda immédiatement cette récompense sans se douter de ce qui allait suivre...

1. Écrivez une fonction qui prend en argument un entier (i.e. le nombre de cases) et qui retourne le nombre total de grains de riz que le roi aurait dû offrir. Testez avec un échiquier de taille 5*5 pour commencer. Affichez le résultat dans le terminal.
2. Testez avec un échiquier complet de 64 cases. Que se passe-t-il à votre avis ? Pour vous en rendre mieux compte, essayez de programmer une boucle qui affiche le résultat pour un nombre de case allant de 1 à 64.

Un test unitaire est un test qui permet de vérifier le bon fonctionnement d'une fonction de votre programme : c'est à destination des développeurs. À chaque modification d'un programme, les tests unitaires associés permettent en effet de détecter de possibles problèmes. Dans plusieurs langages, on peut utiliser une assertion. En C, la fonction `assert(conditionLogique)` permet de tester une condition logique (i.e. quelque chose qui est vrai ou faux) et dans le cas où la condition n'est pas respectée, cela arrête votre programme et vous retourne un message d'erreur. L'avantage, c'est que si vous compilez en mode « Release » (i.e. livrable aux clients), les `assert` ne sont plus pris en compte et n'influencent donc aucunement la performance de votre programme.

3. Avec les observations que vous avez pu faire précédemment, modifiez votre fonction pour qu'elle teste que votre résultat soit dans la plage de valeurs attendue en utilisant un `assert`.

Une façon plus générale de faire des tests unitaires est de programmer plusieurs scénarios d'utilisation d'une fonction. Il existe de nombreuses manières de le faire, différentes selon les langages, parfois plus ou moins adaptées. L'idée est le plus souvent de tester plusieurs valeurs d'entrée (choisies judicieusement) et de vérifier que l'on obtient bien les résultats attendus. Il y aurait beaucoup à dire sur les tests, sachez simplement que cela est de plus en plus central dans le développement d'un logiciel (i.e. certaines méthodes de développement AGILE recommandent de programmer les tests avant même de programmer le système). Pensez à en faire un minimum et à les sauvegarder avec votre programme : cela vous permettra de vérifier rapidement si les dernières modifications que vous avez effectuées n'ont pas eu d'impacts négatifs, ce qui représente un gain de temps pour la suite d'un projet.

4. Écrivez un autre programme (i.e. contenant un autre `main`) qui n'a pour fonction que d'enchaîner vos scénarios de tests.
5. Cherchez une solution pour permettre à votre fonction de pouvoir calculer la valeur pour 60 cases. Exécutez à nouveau votre programme de tests pour vérifier (si vous avez fait des `printf` dans celui-ci, il y aura peut-être une petite modification à effectuer...). Est-ce que cela fonctionne pour 64 cases (i.e. un vrai échiquier) ?

5 Les fonctions récursives (vs. les fonctions itératives)

Toutes les fonctions que vous avez développées jusqu'ici sont pour la plupart ce que l'on appelle des fonctions itératives : ce sont des fonctions qui répètent (ou itèrent) des opérations un certain nombre de fois, le plus souvent en utilisant des boucles.

1. Écrivez une fonction qui calcule le factoriel d'un entier en utilisant une boucle `for`.
2. Écrivez une autre fonction, dite récursive, qui calcule la même chose mais sans utiliser de boucle (indice : cette fonction doit s'appeler elle-même).
3. D'après vous, quels peuvent être les intérêts (ou les défauts) d'une fonction récursive par rapport à une itérative ?

6 BONUS (avec pointeurs) : Initiation au passage de paramètres par référence

Toutes les fonctions que nous avons vues dans ce TP font ce que l'on appelle un "passage de paramètres par valeur". Cela signifie que lorsque l'on a placé une expression en paramètre, une copie de son contenu a été copiée dans une variable locale interne à la fonction. Dans la fonction, aucune modification de cette variable locale n'entraîne de modification de la variable passée en paramètre, car les modifications ne s'appliquent qu'à la copie interne de cette dernière.

1. Pour vous en convaincre, testez le code ci-dessous :

```
void test(int varInt);

int main(int argc, char *argv[]){
        int varInt = 0;
        printf("Initialement, varInt=%i.\n", varInt);
        test(varInt);
        printf("Apres modification, varInt=%i.\n", varInt);
        return 0;
}

void test(int varInt){
        varInt = 3;
}
```

Certains problèmes nécessitent toutefois de pouvoir modifier une variable externe dans une fonction (Exemple : si vous voulez modifier plus d'une variable avec une seule fonction, cela est bien utile car on ne peut avoir qu'un seul `return`). Cela est possible en C, ce que l'on appelle un "passage de paramètres par variable" ou plus spécifiquement un "passage de paramètres par référence".

2. Testez le code ci-dessous pour voir la différence :

```c
void test(int* pointeurSurVarInt); //Notre fonction ne prend plus un entier comme
    parametre, mais un pointeur sur une valeur entiere.

int main(int argc, char *argv[]){
        int varInt = 0;
        printf("Initialement, varInt=%i.\n", varInt);
        int* pointeur = &varInt;//valeur d'un pointeur = adresse d'une variable a
            laquelle on accede via le symbole &
        test(pointeur);//On aurait pu ne pas introduire de pointeur dans le main et
            faire directement test(&varInt);
        printf("Apres modification, varInt=%i.\n", varInt);
        return 0;
}

void test(int* pointeurSurVarInt){
        *pointeurSurVarInt = 3; //Ici le symbole * a un sens different : ca signifie
             qu'on va modifier la valeur pointee par le pointeur.
}
```

Nous aurons le temps de revenir sur les passages de paramètres par référence plus tard. Sachez simplement que ce n'est pas une utilisation anecdotique en C : ils sont souvent plus rapides et plus économes en mémoire (pas de copie interne de variable). De nombreux développeurs utilisent cette méthode par défaut et réservent les `return` à la gestion des erreurs (les fonctions retournent un booléen selon le succès ou non de la fonction). Néanmoins, le passage par référence a une syntaxe plus lourde et les risques d'erreur sont plus élevés (l'inversion entre adresse et valeur - 2 nombres - ne signalera pas d'erreur pour autant par exemple).