

Utilisation de Lua comme langage de script - Partie 1 : Utiliser Lua dans du code C

par [Michel de VERDELHAN](#)

Date de publication : 21/09/2006

Dernière mise à jour :

Dans ce tutoriel, je présente l'utilisation du langage de script Lua dans un programme C. Ce tutoriel n'a pas pour but d'expliquer la syntaxe et la programmation en Lua. L'objectif de ce tutoriel est de montrer comment interfacier Lua et le C pour qu'ils puissent communiquer ensemble.

- I - Pourquoi utiliser un langage de scripts ?
- II - Description de Lua
- III - Utiliser les variables entre Lua et son programme C
 - III-1 - Lire une variable Lua
 - III-2 - Lire une table Lua
 - III-3 - Passer une variable à Lua
- IV - Utiliser des fonctions entre Lua et son programme C
 - IV-1 - Appeler une fonction C depuis Lua
 - IV-2 - Appeler une fonction Lua depuis le C
 - IV-2-A - Vérifier l'existence d'une fonction Lua
 - IV-2-B - Fonction simple
 - IV-2-C - Fonction avec paramètres
 - IV-2-D - Fonction avec valeur de retour
- V - Conclusion

I - Pourquoi utiliser un langage de scripts ?

De nombreux moteurs de jeux utilisent un langage de scripts à plusieurs niveaux du déroulement du programme. On peut se demander quels sont les avantages à utiliser un langage de scripts au lieu de tout programmer directement en C dans son moteur de jeu. Ces avantages sont multiples, en voici quelques uns.

Un problème apparaît quand un projet commence à prendre de l'ampleur : les temps de compilations peuvent devenir de plus en plus importants, allant de plusieurs minutes lorsqu'on ne recompile qu'une partie du projet à plusieurs dizaines de minutes, voire même plusieurs heures pour les très gros projets, lorsqu'on doit recompiler tout le projet. Ces temps de compilation font perdre un temps précieux, surtout lorsqu'on est en phase de mise au point et que le changement d'une variable fait perdre dix minutes... Sur ce point, les langages de scripts ont l'avantage d'être évalué à l'exécution, il n'y a donc pas de temps de compilation, et on gagne donc du temps pour tous les problèmes de mise au point.

Un autre problème est que les développeurs de jeux vidéo ne sont pas tous des programmeurs. Il y a des graphistes, des level designers et autres qui peuvent être appelés à devoir changer le comportement du jeu pour mieux correspondre aux objectifs visés. N'ayant pas forcément de notions avancées de la programmation, il faut pouvoir leur donner la possibilité d'effectuer ces changements sans pour autant savoir ce qu'est un pointeur, une allocation mémoire ou autre. Les langages de scripts ont donc généralement une syntaxe simple et se limitent à des paradigmes de programmation connus, permettant ainsi une programmation plus simple à appréhender. De plus, ces langages gèrent généralement la mémoire eux même avec, par exemple, un ramasse-miettes (garbage collector en anglais) pour le langage que nous allons étudier ici.

Un autre point qui peut pousser à l'utilisation de langages de scripts vient des mods. En effet, on peut vouloir proposer à l'utilisateur de faire ses propres modifications du jeu sans pour autant lui donner tout le code source. Pour cela, on peut mettre en place des systèmes basés sur l'utilisation de bibliothèques dynamiques comme dans les derniers moteurs d'ID software, mais on peut aussi réaliser un système où le moteur du jeu est dirigé par des scripts facilement modifiables par l'utilisateur. L'avantage des scripts par rapport aux DLL est qu'il n'y a pas besoin d'avoir un compilateur sous la main pour pouvoir réaliser son mod. Ceci permet une plus grande ouverture aux utilisateurs.

Malgré ces avantages, l'utilisation de scripts a aussi des inconvénients. Les scripts sont plus lents que du code compilé, ce qui oblige à bien choisir quelles parties du programme seront réalisées en code natif et quelles seront celles réalisées dans le langage de scripts. Ainsi, on réalisera certainement les algorithmes de recherches de chemins en C, mais on donnera la possibilité à l'utilisateur d'exploiter les résultats dans les scripts.

Un autre problème est que les scripts sont généralement stockés en mode texte directement lisible par l'utilisateur. Si ceci permet une modification plus facile, ça permet aussi aux petits malins de changer le comportement du jeu pour réaliser des exploits ou prendre l'avantage durant une partie réseau. Néanmoins, certains langages de scripts permettent de pré- compiler les scripts, comme c'est le cas de celui que nous allons utiliser.

II - Description de Lua

Le langage de scripts que nous allons utiliser est Lua. Ce langage de scripts à été crée en 1991, et a déjà été utilisé dans de nombreux jeux commerciaux comme par exemple dans Far Cry de Crytek. Je ne présente pas plus l'historique de Lua dans ce tutoriel, pour cela, vous pouvez toujours aller sur le site officiel, mais je vais rapidement présenter quelques points importants sur le fonctionnement du langage.

Ce langage a l'avantage de pouvoir utiliser indifféremment des scripts en mode texte ou des scripts compilés. Les scripts compilés en Lua ne sont plus lisibles par l'utilisateur, et permettent un temps de chargement plus rapide, mais, par contre, l'exécution n'est pas plus rapide étant donné que c'est le même moteur de scripts qui tourne par derrière. En fait, la compilation des scripts ne fait que créer la structure du programme directement pour que Lua puisse l'utiliser, et sauvegarde le résultat dans un fichier. Ainsi, lors du chargement du script compilé, Lua n'a plus à faire l'analyse et peut donc immédiatement l'utiliser, ce qui accélère les temps de chargements.

Lua utilise une pile d'exécution virtuelle pour fonctionner. Cette notion est très importante car l'ensemble de l'interfaçage entre Lua et le C passe par cette pile virtuelle. Ceci est intéressant notamment pour le passage de paramètres aux fonctions Lua et pour récupérer les valeurs de retour. Il est important lorsqu'on développe un programme qui utilise Lua de bien faire attention à ce que la pile ne passe pas dans un état incohérent pour éviter d'avoir des comportements indéterminés de l'interpréteur de scripts. En fait c'est la même chose que pour un programme en assembleur : une mauvaise utilisation de la pile peut avoir des conséquences invisibles au départ et donc très difficiles à déboguer par la suite... La technique pour toujours avoir une pile valide est de toujours remettre la pile dans l'état dans lequel on l'a trouvée au début de notre code, ainsi, on revient à chaque fois à un état cohérent.

Il est important de noter que lorsqu'on fait appel à Lua dans notre code C, on utilise intensivement la pile via des indices.

Ces indices sont gérés de deux façons différentes :

- Les indices positifs sont des indices relatifs à la base de la pile.
- Les indices négatifs sont des indices relatifs au sommet de la pile.

Le langage propose aussi un mécanisme d'extension du fonctionnement du langage via un système de meta table. Ce système est relativement complexe, je ne l'expliquerai donc pas ici, mais nous l'utiliserons pour mettre en place un système de programmation objet qui n'est pas supporté de base par Lua. En effet, à la base, Lua est un langage de programmation procédurale, on ne peut donc utiliser que des fonctions comme en C. C'est cette programmation procédurale que je présente dans ce tutoriel.

L'installation de Lua est très simple, elle consiste en un fichier de bibliothèque (.lib sous visual studio, .a sous gcc) et en quelques headers qu'on peut soit copier dans le répertoire include du compilateur, soit mettre dans le projet et signaler au compilateur d'aller les chercher (mais la je vous laisse faire, c'est à vous de savoir configurer votre compilateur ;-)).

Les fichiers nécessaires à l'utilisation de Lua sont fournis dans le fichier zip du tutoriel que vous trouverez en bas de la page.

Pour pouvoir utiliser Lua, il ne nous reste plus qu'à inclure les fichiers lua.h, lualib.h et lauxlib.h si on est en C, et le fichier lua.hpp si on est en C++. Voilà, on peut maintenant commencer à utiliser Lua.

Avant de détailler plus l'interfaçage de Lua avec du C, il faut d'abord savoir comment créer un contexte d'exécution Lua et lancer un script. Ceci se fait comme cela :

Initialisation de Lua

```
lua_State * state;
// on créer un contexte d'exécution de Lua
state = lua_open();
// on charge les bibliothèques standards de Lua
luaL_openlibs(state);
// on lance le script lua
if (luaL_dofile(state, "script.lua")!=0)
{
    // il y a eu une erreur dans le script
    fprintf(stderr, "%s\n", lua_tostring(state, -1));
    exit(0);
}
```

Dans ce code, la variable `state` représente le contexte d'exécution de Lua. Cette variable est très importante car elle sera utilisée lors de chaque appel de fonction Lua pour déterminer sur quel contexte travailler. En effet, on peut très bien avoir plusieurs scripts Lua lancé en même temps avec chacun sa pile d'exécution distincte des autres.

Un autre point important de Lua est que les variables ne sont pas typées. C'est le contenu des variables qui détermine son type. Ainsi, une même variable pourra être coup sur coup un nombre puis une chaîne de caractères. Ceci nous obligera donc dans nos programmes à vérifier que les variables Lua qu'on récupère sont bien du bon type. Il existe en Lua un type particulier : `nil`. Une variable typée `nil` est en fait une variable qui n'a pas de type (donc pas de valeur assignée).

III - Utiliser les variables entre Lua et son programme C

Dans cette partie, je montre comment faire pour récupérer les valeurs de variables Lua en C. Je détaille aussi la lecture des tables Lua depuis le C car ce point est un peu plus complexe qu'une simple lecture de variable.

III-1 - Lire une variable Lua

La lecture d'une variable Lua est très simple. Il suffit de demander à Lua d'empiler le contenu de la variable en connaissant son nom. Lua met la valeur en sommet de pile, il ne nous reste plus qu'à vérifier que la valeur empilée est bien du type souhaité et à récupérer la valeur. Voici par exemple le code nécessaire pour récupérer un nombre.

Code de récupération d'un nombre en Lua depuis le C

```
// récupération d'une variable (number) Lua depuis le code C
lua_settop(state,0);
lua_getglobal(state,"var");
if (lua_isnumber(state,1))
{
    printf("valeur de la variable var : %f\n",lua_tonumber(state,1));
}
lua_pop(state,1);
```

Et voici le code pour récupérer une string.

Code de récupération d'une string en Lua depuis le C

```
// récupération d'une variable (string) Lua depuis le code C
lua_settop(state,0);
lua_getglobal(state,"toto");
if (lua_isstring(state,1))
{
    printf("valeur de la variable toto : %s\n",lua_tostring(state,1));
}
lua_pop(state,1);
```

Il est important de bien dépiler la valeur après l'avoir récupérée, car sinon, on risque de mettre la pile dans un état incohérent.

Comme vous pouvez le voir, ce code est relativement simple et on peut très bien en faire une petite fonction qui va nous faciliter le travail, mais je vous laisse la faire ;-)

III-2 - Lire une table Lua

La lecture de table est un peu plus compliquée que pour une simple variable. En Lua, une table peut être vu comme un tableau indexé par n'importe quel type de variable. Ainsi, on peut très bien indexer une table en même temps avec des entiers et des chaînes de caractères. Du coup, la lecture de la table est plus complexe.

Il faut d'abord demander à Lua d'empiler le tableau en sommet de pile. Ensuite, on empile la valeur de l'indice du tableau qu'on souhaite lire, puis on demande à Lua de remplacer cet indice par sa valeur. On peut ensuite lire la valeur comme une variable classique. Voici un exemple de code pour lire la valeur de la table Lua tableau[2] :

Récupération d'une valeur dans une table Lua depuis le C

```
//récupération du contenu d'un tableau Lua depuis le code C
lua_settop(state,0);
lua_getglobal(state,"tableau");
if (!lua_istable(state,1))
{
    fprintf(stderr,"la variable tableau n'est pas un tableau\n");
}
lua_pop(state,1);
```

Récupération d'une valeur dans une table Lua depuis le C

```
}
else
{
    // on veut adresser la variable d'indice 2
    lua_pushnumber(state,2);
    // maintenant on a dans la pile l'indice à la place 1 et le tableau
    // à la place 2 (car on a empilé l'indice par dessus)
    // on demande à Lua de remplacer le haut de la pile (donc notre indice)
    // par le contenu de la case du tableau en lui donnant l'adresse du tableau
    lua_gettable(state,-2);
    // on vérifie la donnée récupérée
    if (lua_isstring(state,-1))
    {
        printf("valeur recuperé à l'indice 2 : %s\n",lua_tostring(state,-1));
    }
    // on dépile les deux éléments (la valeur de la case et le nom
    // du tableau)
    lua_pop(state,2);
}
```

Encore une fois, on s'assure que la pile retourne dans un état cohérent après la lecture.

III-3 - Passer une variable à Lua

Contrairement à ce qu'on pourrait penser, ici, je ne montre pas la méthode à mettre en oeuvre pour passer une variable C à Lua... En fait, ceci est une mise en garde. Si on peut facilement lire une variable Lua depuis le C et vice versa, il ne faut pas oublier que les deux variables ont une vie différente. La variable C va évoluer dans le contexte de l'application C alors que la variable Lua va évoluer dans le contexte du script Lua... Le passage de variables C vers Lua doit donc être évité autant que possible, et ne doit être utilisé que pour des passages de paramètres aux fonctions Lua.

IV - Utiliser des fonctions entre Lua et son programme C

Bien que le fait de pouvoir lire des variables Lua depuis le C puisse être très pratique, pour utiliser les fichiers de scripts comme des fichiers de configuration par exemple, on se retrouve très vite limité. En effet, avec juste des variables, on ne peut pas faire d'interfaçage complexe entre notre programme et nos scripts. Pour cela, il faut pouvoir utiliser du code C dans Lua et du code Lua dans notre code C. C'est ce que nous allons voir maintenant. Dans un premier temps, je montre comment appeler du code C depuis Lua, puis je montre comment appeler des fonctions Lua depuis notre code C.

IV-1 - Appeler une fonction C depuis Lua

Le fait de pouvoir appeler des fonctions C depuis notre code Lua est ce qui nous permet de diriger notre programme C via Lua. Pour cela, c'est très simple. Il suffit d'avoir une fonction qui a la signature suivante.

Signature d'une fonction C callable depuis Lua

```
int functionName(lua_State* L);
```

Le type de retour doit être un entier qui spécifie le nombre de valeurs de retour (Lua permettant de retourner plusieurs valeurs), et le paramètre de la fonction est le contexte d'exécution dans lequel est appelée la fonction. Voici un exemple de fonction C simple qu'on va appeler depuis Lua :

Fonction helloWorld appelée depuis Lua

```
int helloWorld(lua_State* L)
{
    printf("hello world/n");
    return 0;
}
```

Remarquez que la fonction ne retournant pas de valeurs à Lua, notre fonction C retourne 0.

Pour pouvoir appeler cette fonction, il faut aussi signaler à Lua son existence et lui donner son adresse pour que Lua puisse l'appeler, ce qui se fait comme ceci :

Enregistrement de notre fonction C pour qu'elle puisse être appelée depuis Lua

```
lua_register(state, "helloWorld", helloWorld);
```

Le premier paramètre est le contexte d'exécution du script, le second est le nom de la fonction dans Lua (on peut très bien spécifier un nom différent dans Lua que celui utilisé en C), et le dernier paramètre est la fonction à appeler par Lua.

Maintenant nous pouvons appeler la fonction dans un script Lua, elle est bien reconnue et exécutée.

Nous savons maintenant appeler une fonction Lua, mais nous n'avons pas vu comment récupérer les paramètres de cette fonction. C'est ce que nous allons voir maintenant.

Dans Lua, les paramètres sont passés dans la pile d'exécution, ils sont donc empilés et nous les retrouvons en haut de pile au début de notre fonction C. En fait, lors de l'appel d'une fonction, Lua empile les paramètres dans l'ordre dans lequel ils apparaissent dans le script, puis il déplace la base de la pile pour qu'elle se retrouve au niveau du premier paramètre. Nous avons donc le premier paramètre à l'indice 1 (Lua commence à compter à partir de 1), le second à l'indice 2 etc... Ceci permet de s'assurer qu'une fonction ne peut pas modifier la pile de la fonction qui l'a appelée. Pour récupérer le nombre d'arguments, il faut appeler la fonction `lua_gettop()`, et ensuite, on peut récupérer chaque argument en fonction de sa position relative à la base de la pile grâce au fonction

lua_to* comme illustré dans les fonctions suivantes.

Fonction C affichant une chaîne de caractères passée en paramètre depuis Lua

```
int affichage(lua_State* L)
{
    int nbArguments = lua_gettop(L);
    if (nbArguments != 1)
    {
        fprintf(stderr, "nombre d'arguments invalide");
        return 0;
    }
    printf("parametre : %s\n", lua_tostring(L,1));
    return 0;
}
```

La première fonction prend en paramètre une chaîne de caractères et l'affiche dans la console. On peut remarquer que c'est au programmeur d'effectuer la vérification que les paramètres passés sont valides.

Fonction C affichant tous les paramètres passés par Lua

```
int printParams(lua_State* L)
{
    printf("affichage des paramètres de la fonction : \n");
    int nbParams = lua_gettop(L);
    // Attention : lua commence à compter à partir de 1
    for (int i = 1; i <= nbParams; i++)
    {
        if (lua_isnumber(L,i))
        {
            printf("%f\n", lua_tonumber(L,i));
        }
        else if (lua_isboolean(L,i))
        {
            if (lua_toboolean(L,i))
            {
                printf("true\n");
            }
            else
            {
                printf("false\n");
            }
        }
        else if (lua_isstring(L,i))
        {
            printf("%s\n", lua_tostring(L,i));
        }
    }
    return 0;
}
```

La deuxième fonction affiche aussi les paramètres qui lui sont passés, mais cette fois le nombre de paramètres n'est pas fixé, et leur type non plus.

Notez que les indices dans la pile en Lua commencent à 1 et non pas à 0 comme en C. Pour adresser la base de la pile il faut donc adresser la variable d'indice 1.

Encore une fois, ces fonctions ne retournent pas de valeurs de retour à Lua, elles retournent donc 0. Nous allons maintenant voir comment faire pour retourner des valeurs de retour à Lua avec la fonction suivante.

Cette fonction prend un nombre en entrée et retourne le nombre divisé par deux. Comme vous pouvez le voir, l'envoi de valeurs de retour à Lua est très simple, il suffit d'empiler la valeur en sommet de pile et de signaler que la fonction retourne bien des valeurs de retour en retournant le nombre de valeurs empilées. Voici donc la fameuse fonction :

Fonction montrant l'utilisation de la pile pour envoyer des valeurs de retour à Lua

```
int halfParam(lua_State* L)
```

Fonction montrant l'utilisation de la pile pour envoyer des valeurs de retour à Lua

```

{
    int nbParam = lua_gettop(L);
    if (nbParam != 1)
    {
        printf("halfParam : mauvais nombre de paramètres\n");
        return 0;
    }
    if (!lua_isnumber(L,1))
    {
        printf("halfParam : le paramètre doit etre de type number\n");
        return 0;
    }
    float param = lua_tonumber(L,1);
    float ret = param / 2.0f;
    lua_pushnumber(L,ret);
    return 1;
}

```

Bien entendu, pour que ces fonctions puissent être appelées depuis Lua, il ne faut pas oublier de les enregistrer auprès de Lua comme ceci :

```

lua_register(state, "affiche", affiche);
lua_register(state, "printParams", printParams);
lua_register(state, "halfParam", halfParam);

```

Maintenant que nous savons comment appeler du code C depuis Lua, il nous faut voir comment faire pour appeler du code Lua depuis le C.

IV-2 - Appeler une fonction Lua depuis le C

Bien que la majeure partie de l'interface entre le C et Lua consiste à créer des fonctions C qui seront appelées depuis Lua, il peut être pratique d'appeler des fonctions Lua pour plusieurs raisons :

- Créer une fonction callback qui se déclenche sur un événement donné du moteur de jeu (un ennemi repère le joueur, le joueur appuie sur un bouton...).
- Appeler une fonction qui peut être surchargée par l'utilisateur. Lorsqu'une fonction définie en C est redéfinie dans un script Lua, c'est cette dernière qui sera appelée lors du déroulement du script. On peut donc vouloir appeler depuis le C cette fonction surchargée au lieu d'utiliser directement la fonction C.

Pour utiliser des fonctions Lua, nous verrons d'abord comment vérifier l'existence d'une fonction, puis nous verrons comment appeler une fonction simple, ensuite, je montrerai comment appeler des fonctions avec paramètres, puis les fonctions avec valeurs de retour.

IV-2-A - Vérifier l'existence d'une fonction Lua

Pour pouvoir appeler une fonction Lua depuis notre code C, il faut d'abord vérifier son existence. Pour cela, il faut empiler le nom de la fonction que l'on souhaite appeler, puis faire appel à `lua_isfunction()` avec comme paramètres le contexte d'exécution Lua et l'indice relatif au sommet de la pile ou a été empilé le nom de la fonction à tester. Notez bien ici que l'indice soit un indice négatif. Ceci signifie à Lua que l'indice est relatif au sommet de pile. Dans Lua, les indices positifs sont relatifs à la base de la pile, et les indices négatifs sont relatifs au sommet de la pile. Le code pour vérifier l'existence d'une fonction donne donc ça :

```

// appel d'une fonction Lua depuis le code C
// on empile le nom de la fonction qu'on souhaite lancer
lua_getglobal(state, "funcName");
// on vérifie si la fonction existe bien
if (!lua_isfunction(state, -1))
{

```

```

        // la fonction n'existe pas
        fprintf(stderr,"la fonction funcName n'existe pas\n");
        lua_pop(state,1);
    }
    else
    {
        // code à effectuer pour l'appel de la fonction
    }
}

```

IV-2-B - Fonction simple

Maintenant que nous savons comment faire pour vérifier que nous pouvons bien appeler une fonction, il ne nous reste plus qu'à appeler cette fonction. Pour cela rien de plus simple, il suffit :

- d'empiler le nom de la fonction qu'on souhaite appeler. C'est ce que nous avons fait lors de la vérification de l'existence de la fonction dans Lua.
- d'utiliser la fonction `lua_call`.

La fonction `lua_call` prend en paramètre le contexte d'exécution de Lua, le nombre d'arguments de la fonction et le nombre de valeurs de retour. Dans le cas d'une fonction simple sans paramètres et sans valeurs de retour, le code d'appel complet donne donc :

Appel d'une fonction Lua depuis le C

```

// appel d'une fonction lua depuis le code C
// on empile le nom de la fonction qu'on souhaite lancer
lua_getglobal(state,"helloWorld");
// on vérifie si la fonction existe bien
if (!lua_isfunction(state,-1))
{
    // la fonction n'existe pas
    fprintf(stderr,"la fonction helloWorld n'existe pas\n");
    lua_pop(state,1);
}
else
{
    // on appelle la fonction qui a 0 argument et 0 retour
    lua_call(state,0,0);
}

```

IV-2-C - Fonction avec paramètres

Pour le cas des fonctions avec paramètres, il suffit de rajouter, entre l'empilement du nom de la fonction et l'appel à `lua_call`, l'empilement des paramètres dans l'ordre que nous avons vu plus haut. Il ne faut pas oublier non plus de bien donner le bon nombre de paramètres lors de l'appel à `lua_call`. Voici un exemple simple d'appel de fonction prenant en paramètre une chaîne de caractères :

Appel d'une fonction Lua depuis le code C avec un paramètre

```

// appel d'une fonction lua depuis le code C
// on empile le nom de la fonction qu'on souhaite lancer
lua_getglobal(state,"luaPrint");
// on vérifie si la fonction existe bien
if (!lua_isfunction(state,-1))
{
    // la fonction n'existe pas
    fprintf(stderr,"la fonction luaPrint n'existe pas\n");
    lua_pop(state,1);
}
else
{
    // la fonction existe, on lui passe une chaîne de caractères comme
    // premier argument
    lua_pushstring(state,"le message passé en paramètre");
    // on appelle la fonction qui a 1 argument et 0 retour
}

```

Appel d'une fonction Lua depuis le code C avec un paramètre

```
lua_call(state,1,0);
}
```

IV-2-D - Fonction avec valeur de retour

Maintenant, il ne nous reste plus qu'à savoir récupérer la valeur de retour d'une fonction Lua pour pouvoir l'exploiter dans nos programmes C. Comme pour tout en Lua, il faut passer par la pile. Comme nous l'avons vu, dans la partie sur les retours dans des fonctions C, les valeurs de retour sont empilées en sommet de pile. Pour les récupérer, il nous suffit donc de lire les valeurs en sommet de pile avec un indice négatif, et voilà, nous avons nos valeurs de retour... Voilà un petit exemple d'appel à une fonction qui effectue l'addition entre deux nombres passés en paramètres et retourne le résultat de cette addition :

Appel d'une fonction Lua et récupération de la valeur de retour

```
// on empile le nom de la fonction qu'on souhaite lancer
lua_getglobal(state,"addition");
// on vérifie si la fonction existe bien
if (!lua_isfunction(state,-1))
{
    // la fonction n'existe pas
    fprintf(stderr,"la fonction addition n'existe pas\n");
    lua_pop(state,1);
}
else
{
    // la fonction existe, on lui passe une chaine de caractères comme
    // premier argument
    lua_pushnumber(state,10);
    lua_pushnumber(state,20);
    // on appelle la fonction qui a 2 argument et 1 retour
    lua_call(state,2,1);
    // on récupère la valeur de retour
    float retour = (float)lua_tonumber(state,-1);
    printf("valeur de retour : %f\n",retour);
}
```

V - Conclusion

Comme vous pouvez le voir, l'utilisation de Lua en C est relativement simple. Nous avons maintenant tous les outils en mains pour pouvoir faire communiquer ces deux langages sans difficultés. Néanmoins, nous ne pouvons pas utiliser de classes en Lua dans l'état actuel des choses. Pour cela, il nous faut utiliser les meta tables de Lua. C'est ce que nous verrons dans le prochain tutoriel sur Lua.

Vous pouvez télécharger les sources de ce tutoriel [ici](#) ou [ici \[http\]](#)

Une version PDF de ce tutoriel est disponible [ici](#) ou [ici \[http\]](#)