Trainer : Pour les pas trop nuls

Préambule

Ce tutorial n'a pas été écrit dans le but de devenir une référence, loin de là. Il a été écrit en vu d'être publié sur le site TrainersCity, le tout premier site de trainer en français sur lequel je suis tombé étant plus jeune. Ce tutorial est donc destiné à enrichir ce fabuleux site (un peu austère et qui devrait changer un peu de design d'après moi☺).

De même, ce tutorial en français va permettre, je l'espère à faire développer la scène de la création de trainer en français, sachant qu'il n'y a pas beaucoup de personnes ou groupes français qui en distribue. De ce côté-là je ne saurai dire pourquoi, peut être moins intéressant que de faire des crack/patch/hack ?

D'un autre côté ce tutorial sera écrit de manière simple parce que je n'aime pas faire de longues phrases qui m'embrouilleraient et sera de plus beaucoup plus lisible.

Les étapes de la création d'un trainer, de la recherche des modifications à effectuer jusqu'à la génération du trainer (l'exécutable) sera traité ici.

Ce tutorial s'adresse aux personnes qui ont les bases de l'informatique, savoir utiliser un programme, connaître un tant soit peu l'hexadécimal, le binaire, etc. Il n'y aura donc pas de rappel de ce côté-là et vous pouvez chercher sur le net les informations qui vous aideront.

L'assembleur y sera traité de manière simpliste parce que moi-même je n'y connais pas grand-chose en matière d'assembleur x86 mais de toutes manières les op codes (grosso modo les instructions assembleurs) sont quasiment les mêmes que pour les autres processeurs. Je vous renvoie donc à de la documentation sur le net pour le reste des op codes que vous pourriez rencontrer.

Concernant la programmation à proprement parlé du trainer, celle-ci sera effectuée sous Visual C++. Vous devrez donc avoir les bases de la programmation en C pour vous y retrouver. Je détaillerai quand même les APIs indispensable pour la manipulation des adresses ainsi que le fonctionnement des messages Windows.

Pour la référence pour les recherches de valeurs, j'ai choisi un jeu, certes vieux, mais que tout le monde peut avoir pour suivre ce tutorial : « Virtua Cop 2 », jeu que vous pourrez trouver sur le net.
Pour le logiciel de recherche, ce sera T-Search. Des indications seront données au cours de ce tutorial quant à son utilisation.

Sur ces bonnes paroles, je vous souhaite une bonne lecture.

Qu’est-ce qu’un trainer ?

Tout d’abord, qu’est ce qu’un trainer ?

Un trainer est un petit programme (généralement quelques kilo-octets) qui permet de tricher dans les jeux.

Par exemple dans notre jeu de référence, vous n’aimez pas recharger ? Qu’à cela ne tienne, avec un trainer vous gelez constamment le nombre de balles que vous avez ! N’est ce pas génial ? Bon, pour ceux qui n’aiment pas tricher je comprendrai leur avis, mais ce n’est pas le cas traité ici.

Voilà ce qu’on peut dire sur un trainer. Je vous l’avais dit, ce tutorial fera dans le très concis.

Préparatifs

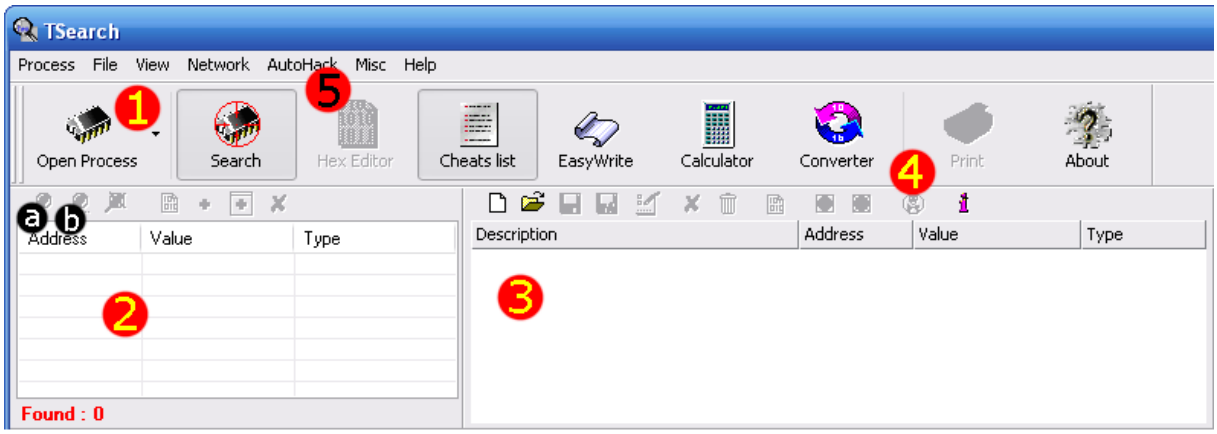
Bon, pour commencer, sachez qu'il vous faudra :

1. Le jeu Virtua Cop 2
2. Un scanneur de mémoire assez avancé (du type T-Search fera l'affaire)
3. Visual C++ (ou tout autre langage si vous connaissez les APIs correspondantes pour ce langage)

Voilà, vous avez tout l'attirail pour créer un trainer.

La recherche de valeurs et d'adresses

Etape ennuyante et parfois longue… mais nécessaire néanmoins.
Donc pour commencer, démarrer le jeu si ce n'est déjà fait et revenez sous Windows (sans quitter le jeu bien entendu).
Démarrer T-Search et ouvrez le processus du jeu (cliquez sur le bouton « Open Process » (1)).



1 : Ouvre un processus (le jeu)
2 : Affiche les adresses trouvées
  a : Nouvelle recherche
  b : Poursuivre la recherche
3 : Sauvegarde les adresses trouvées
4 : Activer Debugger (instructions assembleur)
5 : Debugger (AutoHack)

Servez-vous de cette capture pour vous y retrouver par la suite.

> *Ceci fait, je vous conseil dès à présent d'activer le debugger dans le menu AutoHack (4). Ce debugger permettra par la suite de connaître les instructions assembleur qui accèdent aux adresses trouvées.*

Ceci fait, retournez dans le jeu et comptez le nombre de balles que vous avez. Vous devriez en avoir 6, comme moi. Bien, Revenez sur T-Search et débutez une nouvelle recherche (a).

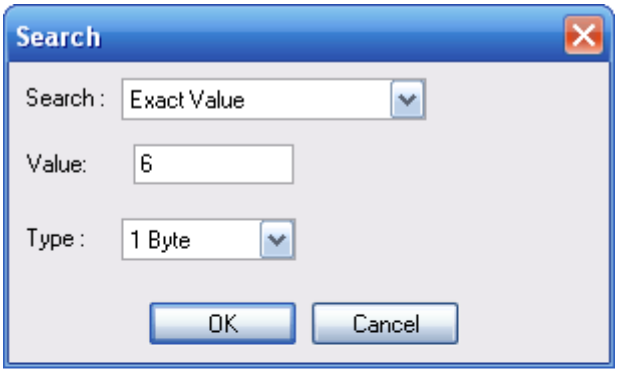
> *Je déconseille la recherche d'une valeur valant 0 ou même 1, valeurs qui sont très souvent trop présent dans un programme. Essayez autant que possible dans vos recherches, d'avoir des valeurs pas trop courantes.*

Puisque vous connaissez également la taille des types d'entier (char, short, int) vous choisirez donc char.

Pour rappel, « char » prend 1 octet, « short » 2 octets, « int » 4 octets et « float » 4 octets ; « char » allant de 0 à 255, « short » de 0 à 65535 et « int » allant de 0 à 4294967295. Le format des floats est un peu spécial en écriture assembleur (mantisse et exposant). Cela n'a pas d'importance ici, on cherche rarement de telles valeurs en hexadécimale. On préfèrera utiliser des recherches par intervalles.

La plupart du temps, int sera quand même conseillée, vu que généralement les développeurs ne se prennent pas la tête à caler au mieux les types correspondants. Vous pourrez donc trouver même avec un int.

Ok, donc cherchez la valeur 6, avec char, short ou int, comme vous voulez et validez.

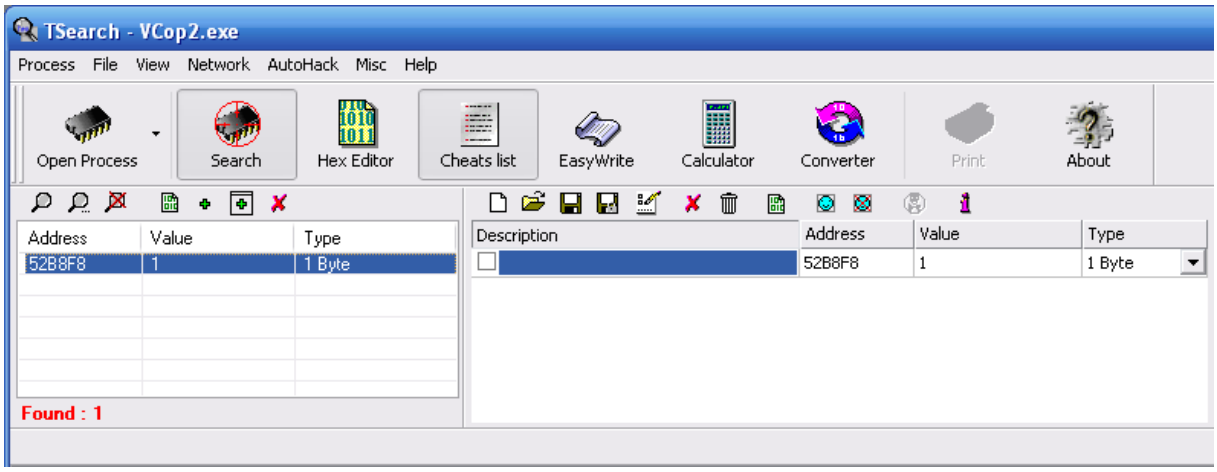


Search : type de recherche
- Exact Value (valeur exacte)
- Range (intervalle)
- Unknown Value (inconnue)

Type : taille mémoire de la valeur à chercher

Voilà, vous aurez plein d'adresses. Retournez dans le jeu et tirer une balle et recommencez la recherche avec l'autre bouton (« Poursuivre la recherche » (b)). Continuez jusqu'à ne trouver que quelques adresses.

Ne pas oublier qu'il faut cliquer sur la loupe mais avec des pointillés en bas pour continuer une recherche et non pas la première loupe qui elle sert à effectuer une nouvelle recherche.



Normalement vous n'aurez plus qu'une seule adresse qui affichera le nombre de balles que vous avez actuellement. Ici il ne me restait plus qu'une balle.

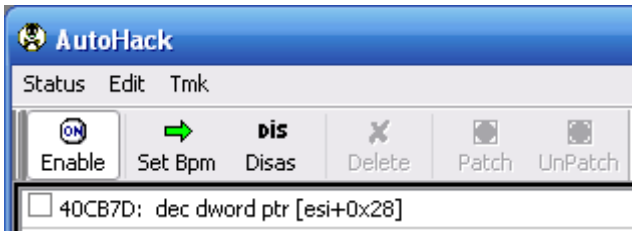
Double cliquez sur cette ligne. Elle apparaîtra sur la droite comme sur cette capture. Cochez la case en dessous de Description pour geler cette valeur (vous pouvez lui en attribuer une autre, pour cela, double cliquez sur la valeur dans la fenêtre de droite toujours). Vous devriez voir une petite tête bleue indiquant que la valeur est gelée. Mettez par exemple 6, comme la valeur initiale. Retournez dans le jeu et « Oh ! Qu'est-ce qu'il se passe ? Je tire ! Oh ! Il reste encore 6 balles ! » Bref, voilà, vous savez maintenant chercher et geler une valeur dans T-Search.

> *Concernant les autres types de recherche, dans un jeu vous serez peut être amené à chercher des valeurs flottantes (décimales), dans ce cas vous devrez chercher par valeurs inconnues si vous ne savez pas la valeur de départ ou bien par intervalle si vous connaissez à peu près la valeur. Dans ce cas entrez les extrémums correspondants.*

On va maintenant passer à l'assembleur.

L'assembleur

Activez le debugger. Pour cela, cliquez sur « AutoHack » (6) et cliquez sur « Enable Debugger ». Cliquez ensuite sur « AutoHack window » pour afficher une nouvelle fenêtre. Décochez la petite tête bleue qui gèle la valeur, on va tracer l'instruction qui enlève une balle à chaque fois qu'on tire. Ceci fait, sélectionnez la ligne contenant l'adresse où est stockée le nombre de balle et cliquez sur la petite tête jaune (4). Retournez dans le jeu et tirez une balle puis revenez sous T-Search. Vous voyez qu'une ligne est apparue, c'est cette instruction qui enlève une balle lorsque vous tirez.



Que signifie cette ligne ?
Le premier bloc est l'adresse où est située cette instruction, ici à l'adresse « 0x0040CB7D ».
Que fait cette instruction ?
« dec dword ptr [esi+0x28] » décrémente la valeur pointée par l'adresse « esi+0x28 ».

Dans notre cas, on n'a pas nécessairement besoin de connaître à quoi correspond cette adresse (on la connaît forcément ici : « esi+0x28 » correspond à l'adresse trouvée pendant la recherche précédente).

Cliquez sur cette ligne. La fenêtre d'en bas va vous afficher le contenu de cette zone mémoire avec les instructions assembleur qui suivent.

Faites un clique droit sur cette ligne (fenêtre d'en bas) et cliquez sur « Nop this line ». L'instruction NOP en assembleur est une instruction qui ne fait rien. En modifiant cette instruction en instruction qui ne fait rien, on fait comme si la décrémentation du nombre de balles ne se faisait pas. Du coup, on ne perdra jamais de balles !

Avec un peu de bon sens et de l'observation et un peu de connaissance en assembleur, vous verrez que vers le bas il y a une instruction qui compare le nombre de balles restantes avec 0, sûrement pour savoir si vous êtes en manque de munitions ou non ou alors afficher « Reload ».

L'instruction suivante effectue un saut à une autre adresse si le test est négatif. Si vous n'avez pas encore changé l'instruction qui décrémente le nombre de balles, amusez vous à changer cette instruction de saut

conditionnel (conditionnel parce que le saut dépend d'un test) par un saut inconditionnel (parce que le saut s'effectuera quelque soit les conditions) : faites clique droit sur l'instruction à changer et cliquez sur « Assemble ». Ceci fait, remplacez JNZ par JMP.

Vous verrez donc que le message « Reload » aura disparu en effectuant un saut inconditionnel. Aucune utilité ici mais c'était pour vous inviter à aller plus loin que ce que vous avez sous les yeux. N'hésitez donc pas à parcourir les instructions assembleur qui sont aux alentours de l'adresse que vous avez trouvé. Vous verrez d'ailleurs qu'il y a une instruction intéressante vers le bas.

Voilà, maintenant vous savez modifier et tracer les instructions assembleur qui modifient les valeurs en mémoire.

Je le répète, pour l'assembleur vous pouvez aller voir sur le net. Vous verrez que vous pourrez faire de jolies choses en connaissant cela.

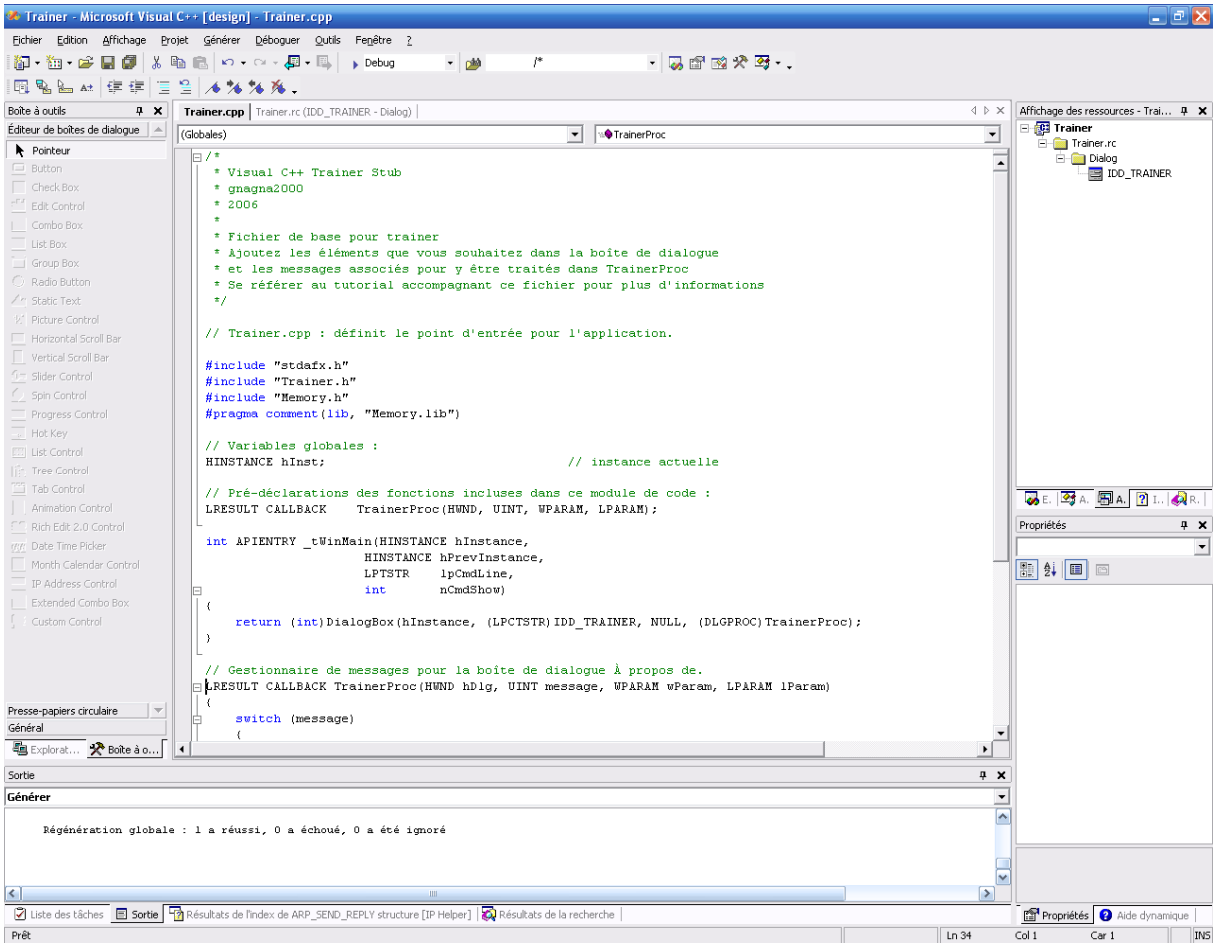
Un dernier mot concernant l'assembleur. La plupart des jeux contiennent des espaces mémoires non utilisées. Ces zones sont appelées « Code Cave » et permettent d'ajouter son propre code assembleur. Par exemple dans l'instruction précédente, on pourrait remplacer le code qui décrémente par un saut à une adresse non utilisé où on a ajouté une instruction qui met toujours 6 dans l'adresse contenant le nombre de balle puis un saut vers l'instruction qui suivait l'instruction de décrémentation. Cette méthode est plus communément appelé « Code Injection ».

Vous savez maintenant effectuer une recherche d'adresse par valeur, tracer l'instruction qui accède à cette adresse et modifier le comportement de celle-ci. Vous avez tout ce qu'il vous faut pour faire votre premier trainer sous Visual C++.

Premier trainer

Le code

Vous vous rappelez l'adresse qui contient le nombre de balles ? 0x0052B8F8 normalement.
Bien pour notre premier trainer on va faire en sorte qu''il nous gèle la valeur 6 à cette adresse.
Ouvrez le fichier exemple joint à ce tutorial sous Visual C++.



La partie de droite correspond à l'explorateur de solution (tous les fichiers composant le projet) et la liste des ressources de votre projet, ici la fenêtre principale).
Au centre, le code source, ici le fichier principal Trainer.cpp.

Pour faciliter les choses, j'ai écrit une librairie statique (donc inutile à transmettre lorsque vous voudrez distribuer vos trainers) qui effectue les tâches de manipulation mémoire à votre place. Vous n'utiliserez donc que les classes associées à cette librairie.

Bien, commençons par écrire notre premier bout de code.
Dans la partie « Variables globales », on va déclarer un pointeur de la classe CheatVAL : Balle. Celle-ci aura en charge de geler une adresse avec une valeur donnée.
On a donc ceci d'écrit en plus :

```
static CheatVAL * Balle;
```

CheatVAL est la classe qui permet de geler une certaine valeur à une certaine adresse et à un intervalle donné.

Dans le « main » du programme (_tWinMain) on va maintenant initialiser cette librairie pour qu'elle puisse connaître dans quel processus écrire. Pour trouver le jeu, on lui passe en paramètre le titre de la fenêtre du jeu, ici : « VirtuaCop 2 ».

```
CHEAT_INIT("VirtuaCop 2");
```

Cette fonction retourne TRUE si elle arrive à trouver le jeu et FALSE dans le cas contraire.
On va maintenant instancier la classe Balle :

```
Balle = new CheatVAL(0x0052B8F8, 6, 100);
```

Ici on souhaite écrire la valeur 6 à l'adresse 0x0052B8F8 toute les 100 ms.
De part la nature de cette classe que j'ai définit, elle est active dès le début de la création. Avec les méthodes associées à cette classe, on peut mettre en pause, redémarrer et vérifier l'état de son exécution (activé ou désactivé).

```
Balle->IsRunning();
Balle->Suspend();
Balle->Resume();
```

- IsRunning() retourne TRUE si cette classe est active et FALSE sinon.
- Suspend() permet de mettre pause cette classe : elle ne mettra plus à jour la valeur.
- Resume() permet de réactiver la classe pour qu'elle mette à jour la valeur.

Vous savez maintenant comment geler la valeur d'une adresse.

Comme on l'a vu précédemment, on a trouvé l'adresse de l'instruction qui permet de décrémenter le nombre de balles. On doit donc changer cette instruction pour qu'elle ne décrémente plus le nombre de balles. Pour cela, on va utiliser la classe CheatHEX qui va remplacer l'instruction par celle qu'on aura définit.

Toujours dans les variables globales, on va déclarer un pointeur, Balle2, de la classe CheatHEX.

```
static CheatHEX * Balle2;
```

Dans le « main », on va devoir instancier cette classe. Avant de faire cela, on va créer deux tableaux de caractères contenant en hexadécimale l'instruction assembleur.

Reprenons l'instruction précédente :
« dec dword ptr [esi+28] »
Cette instruction, si vous regardez dans la fenêtre du bas (« AutoHack Window »), vous verrez qu'il y a sa représentation sous forme hexadécimale : « FF 4E 28 ». On va alors créer un tableau de caractères contenant ces 3 valeurs et un autre tableau contenant la valeur hexadécimale de l'instruction NOP qui correspond à « 90 » en hexadécimale. Il nous faudra alors changer ces 3 valeurs par 3 « 90 ». En notation hexadécimale, on précède toujours par « 0x ». Par exemple, on n'écrit pas « 90 » mais « 0x90 ».

```
char Balle2INIT[] = {0xFF, 0x4E, 0x28};
char Balle2CHANGE[] = {0x90, 0x90, 0x90};
```

Voilà la forme que j'ai donnée ici, vous pouvez bien sûr lui donner n'importe quel autre nom.

Bien, on peut maintenant instancier notre classe Balle2.

```
Balle2 = new CheatHEX(0x0040CB7D, Balle2INIT, Balle2CHANGE, 3);
```

Ceci correspond à : à l'adresse 0x0040CB7D j'ai comme valeurs initiales Balle2INIT et comme valeur modifiées Balle2CHANGE ayant une taille de 3 octets.

Cette classe contient 4 méthodes pour être manipuler :

```
Balle2->Activate();
Balle2->Default();
Balle2->IsActivated();
Balle2->Toggle();
```

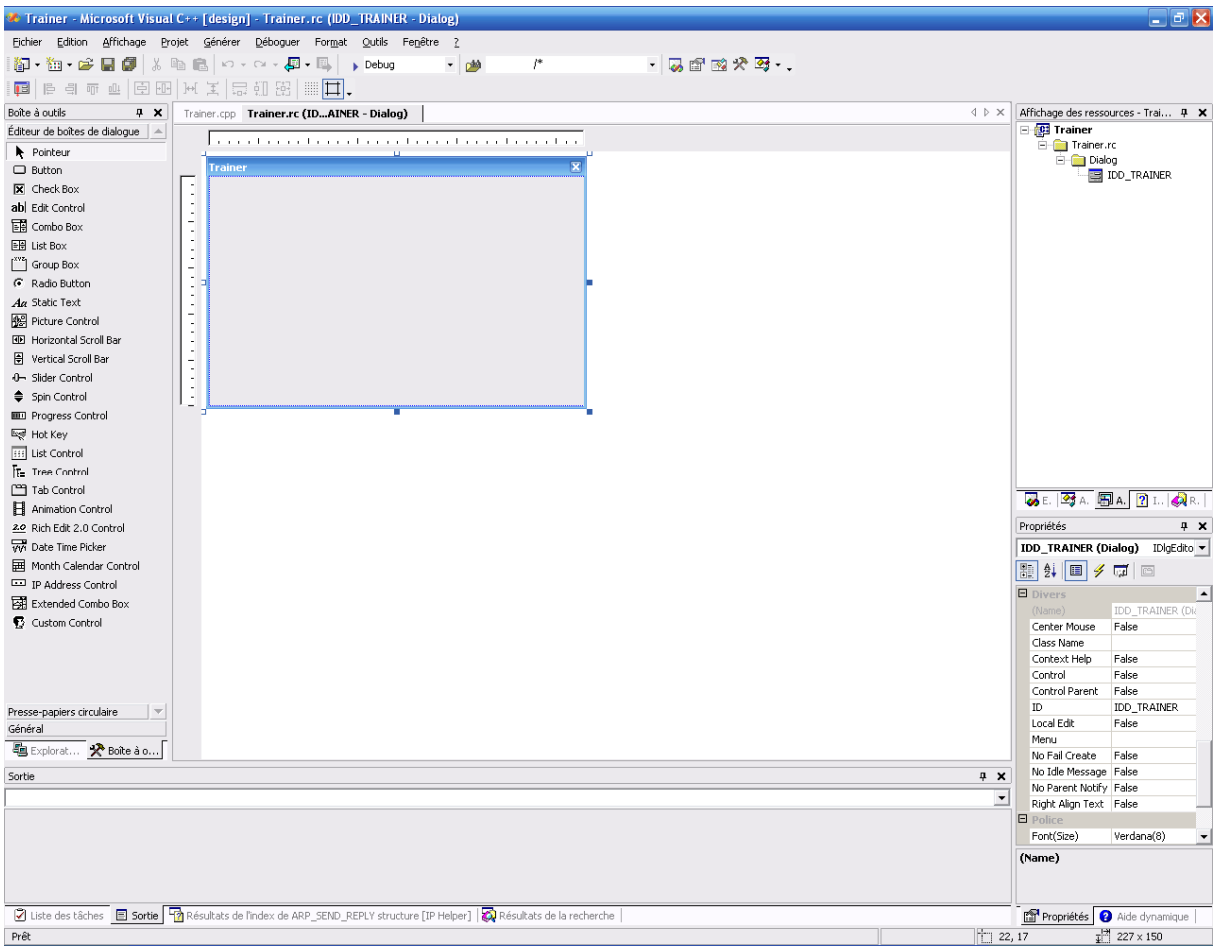
- Activate() : quelque soit l'état de cette classe (activé ou non), on modifie la mémoire par les valeurs contenues dans Balle2CHANGE.
- Default() : l'inverse du précédent, celle-ci remet par défaut l'état de la mémoire (Balle2INIT).
- IsActivated() : retourne TRUE si la classe est active, FALSE sinon.
- Toggle() : permet d'activer et désactiver la classe (passe en mode activé si la classe est désactivé et vice versa).

Voilà, vous savez maintenant comment changer les instructions assembleur !

Passons à la création de l'interface graphique.

Interface graphique

Le moment le plus attendu peut être ? Bon je ne suis pas graphiste et je ne sais pas faire de belles choses. Je ne vous apprendrai donc rien de ce côté-là. Dans la fenêtre des ressources (partie de droite) déroulez l'arborescence jusqu'à trouver IDD_TRAINER. Ceci est l'identifiant de la boîte de dialogue qu'on va utiliser. Double cliquez dessus et la fenêtre devrait s'afficher dans la partie centrale.



Dans la partie gauche, vous verrez des éléments que vous pouvez utiliser : boutons, boîtes d'éditions, barres de défilement, boîte statique, etc. Sélectionnez le « Check Box » et placez-le sur la boîte de dialogue en cliquant dessus. L'élément sera affiché et dans la partie inférieur droite vous pourrez modifier ses propriétés : le texte, s'il est visible ou non, activé ou pas, etc. Retenez bien son identifiant, normalement IDC_CHECK1. On va en avoir besoin lors de l'écriture du code. Retournez

dans l'édition de code source. Dans la fonction « TrainerProc », vous verrez un « switch ». « Switch » permet de tester la valeur qu'on lui passe en paramètre.

```
switch (message)
```

Il est suivit par un bloc (entre accolades) contenant les valeurs qu'on souhaite capturer.

```
case WM_INITDIALOG
case WM_COMMAND
```

On indique donc à Windows qu'on souhaite capturer les messages WM_INITDIALOG et WM_COMMAND.
Pour plus d'information sur les différents messages Windows, reportez vous à la documentation Windows (MSDN, Platform SDK).

Le fonctionnement de Windows est grosso modo celui là : la fonction TrainerProc est appelée à chaque fois qu'un message arrive. Ses paramètres d'appels sont :

```
HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam
```

- hDlg est un « Handle Window », celui qui définit la boîte de dialogue en gros.
- message est le message reçu.
- wParam et lParam sont des paramètres dépendant du message reçu.

Concernant les messages, on se focalisera sur WM_COMMAND, qui est le message que l'on reçoit lorsqu'une commande/action est arrivée.
Quand on reçoit ce type de message, les 16 premiers bits de wParam contiennent l'identifiant de l'élément de la boîte de dialogue en question.

```
case WM_COMMAND:
if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
{
      EndDialog(hDlg, LOWORD(wParam));
      return TRUE;
}
break;
```

Pour cela, on utilisera la macro LOWORD qui permet de récupérer ces valeurs là exclusivement. IDOK et IDCANCEL sont prédéfinit est correspondent à la fermeture de la fenêtre par défaut (on en fait ce qu'on veut de toute façon). Si on reçoit ce genre de message, alors on Ferme la boîte de dialogue avec l'appel de « EndDialog » comme on peut le voir dans l'exemple.

A chaque fin de bloque d'un « case » correspondant à un cas de la valeur d'un « switch », on veillera bien à utiliser le mot clé « break » qui permet d'arrêter la comparaison des valeurs. Sinon les instructions suivantes seront exécutées.

On va donc maintenant ajouter notre propre identifiant pour le « Check Box ».
Après le bloc du « if », on va écrire :

```
if (LOWORD(wParam) == IDC_CHECK1)
{
}
```

On interceptera alors quand une action aura été effectuée sur notre « Check Box ». Entre les accolades, on va vérifier si le « Check Box » est coché ou non.

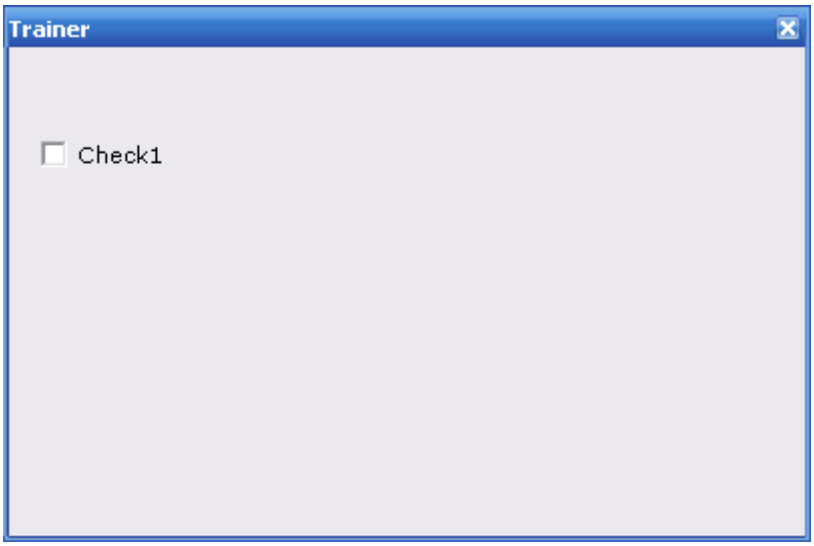
```
if (IsDlgButtonChecked(hDlg, IDC_CHECK1) == BST_CHECKED)
```

Cette macro permet de connaître l'état d'un bouton (un « Check Box » est un bouton aux yeux de Microsoft). Cette fonction renvoie BST_CHECKED s'il est coché et BST_UNCHECKED s'il n'est pas coché. Donc s'il est coché, on va activer notre cheat !

```
if (IsDlgButtonChecked(hDlg, IDC_CHECK1) == BST_CHECKED)
Balle2->Activate();
else
Balle2->Default();
```

Voilà, c'est fini ! Il ne reste plus qu'à embellir le trainer à votre guise et à générer l'exécutable.

Sélectionnez « Release » dans la barre en haut (à la place de « Debug ») et générez le (menu Générer>Générer la solution).



Les APIs utiles

Je vais résumer ici les principales APIs utiles (en association avec ma bibliothèque).

- Pour initialiser la bibliothèque :
```
CHEAT_INIT("Titre de la fenêtre du jeu");
```

- Pour geler une valeur à une adresse :
```
CheatVAL * cheat = new CheatVAL(
                                adresse,
                                valeur,
                                intervalle de temps);
```

- Pour modifier une instruction assembleur :
```
cheat = new CheatHEX(
             adresse,
             tableau de caractères,
             tableau de caractères,
             taille du tableau);
```

Les APIs Win32 :

- Pour vérifier l'état d'un « Check Box » :
```
IsDlgButtonChecked(handle window, identifiant check box);
```

- Pour cocher ou décocher un « Check Box » :
```
CheckDlgButton(handle window, identifiant de l'élément, coché ou non);
```
Prenant comme valeur BST_CHECKED (coché) ou BST_UNCHECKED (non coché)

- Pour récupérer le « Handle » d'un élément de la boîte de dialogue :
```
GetDlgItem(handle window, identifiant de l'élément);
```

- Pour obtenir le nombre d'une boîte d'édition :
```
GetDlgItemInt(handle window, identifiant élément, NULL, signé ou non);
```
On pourra utiliser TRUE ou FALSE pour récupérer un nombre signé ou non (signé signifie pouvant être négatif).

- Pour écrire un nombre dans un élément :
```
SetDlgItemInt(handle window, identifiant élément, valeur, signé ou non);
```

- Pour récupérer le texte d'un élément :
```
GetDlgItemText(
        handle window,
        identifiant de l'élément,
        buffer,
        nombre de caractères max);
```
Le « buffer » est l'adresse où sera stocké le texte obtenu. Un pointeur de chaîne de caractères (`char *`) fait l'affaire.

- Pour écrire un texte dans un élément :

```
SetDlgItemText(handle window, identifiant élément, buffer);
```

« Buffer » est un pointeur vers une chaîne de caractères contenant le texte.

Conclusion

Voilà, vous avez tout ce qu'il faut pour pouvoir créer des trainers de bases. Faites des essaies, consultez la documentation et vous progresserez. N'hésitez pas consulter des forums de programmations Windows pour parfaire vos connaissances et les appliquer ici.
Dans tous les cas, j'espère ne pas avoir été trop concis dans ce que j'ai écrit et que vous aurez compris. J'espère maintenant voir de nouveaux trainers en français apparaître dans ces quelques prochaines années. ☺