TP - API Windows

Exercice 1 - Premiers pas ...

A. Créer sous Visual 9 (2008) un nouveau projet **Test** de type Win32 (pas console et pas vide !). Compiler, exécuter puis analyser les fichiers - source générés par le Wizard. Identifiez les moments - clés dans l'évolution du programme comme l'enregistrement d'une classe de fenêtres, la création d'une fenêtre, la boucle de messages, la ou les procédure(s) de traitement de messages.

- Quel est le style de la classe des fenêtres ?
- Quelle est la couleur du fond de la fenêtre ? Remplacer-la par une couleur violette en utilisant l'API **::CreateSolidBrush** .
- Quels sont les styles et les styles étendus de la fenêtre ?
- Quelle est la taille initiale de la fenêtre ?
- Quels sont les messages traités explicitement au niveau de la fenêtre principale ?
- Est-ce que le code source est écrit en Unicode ou ANSI ? Vérifier !

On souhaite réécrire le programme précédent en utilisant une approche objet en vue de la réutilisation et l'enrichissement du code initial. Notre souhait est de ne pas utiliser de variables globales : si cela est nécessaire, il vaut mieux des membres statiques gérés par des classes. On souhaite une utilisation souple des objets, où l'utilisateur peut choisir le comportement et l'apparence des fenêtres sans modifier le code. Créer un nouveau projet **Exo1** de type Win32 vide. Le code sera écrit en mode compatible ANSI/Unicode. Placer les classes dans un fichier d'entête **Wind.h** (protégé à la double inclusion) et les tests dans **main.cpp** .

B. Modéliser une classe de fenêtres par la classe **CClsWind**. Elle garde comme seul attribut un **ATOM** (pourquoi est-il suffisant ?). Prévoir 3 façons différentes de création :

- En renseignant un maximum de paramètres : nom de la classe, la **WNDPROC**, style de la classe, couleur de fond, curseur (pas de *handle* d'instance, utilisez **GetModuleHandle** à la place). Prévoir des valeurs par défaut raisonnables pour chaque paramètre, sauf pour le nom de la classe qui sera obligatoire.
- En renseignant un maximum de paramètres sans le nom de la classe : pareil, mais on crée un nom unique de classe dans le constructeur, en utilisant le *timestamp* de l'appel (**QueryPerformanceCounter** pour le *timestamp* et **wsprintf** pour le formatage).
- En ne renseignant rien : on ne fait pas d'enregistrement de classe. Comment mémorise-t-on cet état au niveau de l'**ATOM**, pour éviter l'effacement de la classe de fenêtres dans le destructeur ?

Prévoir un destructeur qui désenregistre (si nécessaire !) la classe, un opérateur de *typecast* pour convertir un objet de cette classe vers une chaîne **LPCTSTR** (convertir brutalement l'ATOM vers la chaîne constante car cet opérateur servira uniquement pour les appels des API Windows qui demanderont le nom de la classe) et une méthode **IsValid** qui retourne **true** si la classe est bien enregistrée.

C. Modéliser une fenêtre par la classe **CWind** qui encapsule le *handle* **m_hWnd** de type **HWND** de la fenêtre. Chacun des ses constructeurs va créer la fenêtre en utilisant **CreateEx** avec les paramètres passés par l'utilisateur ou à partir des valeurs par défaut, ou "fabriqués" en interne.

Ecrire un 1er constructeur (pour des fenêtres principales) avec les paramètres suivants :

- le titre de la fenêtre (par défaut une chaîne vide "")
- la position et la taille, regroupées dans une structure **RECT**, passée par adresse (valeur nulle par défaut). Prévoir un membre statique **DefRect** de type **RECT** qui aura une position et une taille par défaut, à utiliser à la place de l'argument, quand l'adresse passée est nulle.

- le style (par défaut le style **WS_OVERLAPPEDWINDOW** et **WS_VISIBLE** défini comme la constante **DEF_STYLE**) et le style étendu de la fenêtre (par défaut nul)
- le nom d'une classe de fenêtres (par défaut pointeur nul : dans ce cas utiliser un attribut statique **DefClsWind** de type **CClsWnd** créé avec les bons paramètres : titre "DefClsName", méthode statique **CWind::WndProc**, fond blanc, bouton de fermeture).
- un ID de menu de type **WORD**, par défaut nul, à utiliser pour charger la ressource d'un menu avec **LoadMenu** (si l'ID n'est pas nul) et la passer à la création de la fenêtre

Prévoir un 2ème constructeur (pour des fenêtres enfants) qui reçoit par adresse un objet de même type que lui (**CWind**) représentant une fenêtre parent, puis un ID d'enfant de type **WORD**, puis les autres paramètres (rectangle, style, style étendu, nom de classe de fenêtres) avec des valeurs par défaut. A la création de la fenêtre on passe le *handle* du parent s'il existe (ou 0 en cas contraire) et l'ID d'enfant comme **HMENU** (*typecast* sauvage !).

Prévoir un 3ème constructeur qui reçoit la couleur de fond, le titre de fenêtre, son rectangle, le style, le style étendu et un style de classe. Il construit l'attribut **LocClsWind** de type classe de fenêtre en utilisant la **CWind::WndProc**, le style de la classe et la couleur de fond. Puis il l'utilise à la création de la fenêtre.

Le destructeur teste que le *handle* n'est pas nul et que la fenêtre existe bien à l'aide de l'API **IsWindow** avant de la détruire à l'aide de **DestroyWindow**.

Prévoir la méthode statique publique **WndProc** de type **WNDPROC** qui va recevoir les messages, traite le cas de **WM_DESTROY** et renvoie les autres messages vers la **DefWindowProc**. L'adresse de cette méthode est utilisée à la création des attributs **DefClsWind** et **LocClsWind**.

Prévoir une autre méthode statique (pourquoi statique ?) **MsgLoop** qui lit et *dispatche* les messages. Comment s'assurer que la fermeture de la fenêtre met fin au programme ?

Rajouter un getter pour **m_hWnd** et des getters/setters pour les styles normaux et étendus.

Testez avec une fenêtre créée avec le 1er constructeur (paramètres par défaut, puis rajouter des paramètres dans l'ordre : un titre, une position en coin supérieur gauche de taille 200 par 200, style par défaut et ascenseur vertical, style étendu "la plus en avant", une classe de fenêtre créée comme variable locale de fond magenta, l'ID **IDR_MENU1** d'un menu créé comme ressource et inclus à partir de "**resource.h**"). Testez avec le 3ème constructeur une fenêtre de fond jaune.

D. Rajouter une console à votre programme GUI. Même si l'OS Windows distingue 2 sous-systèmes (console et GUI), grâces aux API Windows, un programme GUI peut avoir une console (et une seule !) et inversement, un programme console peut créer des fenêtres et travailler avec des messages. Dans notre cas on va utiliser les API **AllocCosole** et **FreeConsole** encapsulées dans une classe **CConsole** avec une instance globale unique :

```
#include <stdio.h>
class CConsole
{
public:
  CConsole() {::AllocConsole();_tfreopen(_T("CONOUT$"),_T("wt"),stdout);}
  ~CConsole() { fclose(stdout); ::FreeConsole(); }
} globConsole;
```

La fonction **freopen** redirige la sortie **stdout** vers la console nouvellement créée, alors que par défaut elle est ignorée.

Tester la console en affichant les IDs des messages dispatchés et ceux reçus par **WndProc**. Y-a-t il une différence ? Pourquoi ? Tester aussi pour une fenêtre créée à partir d'une classe de fenêtre en variable locale qui utilise comme **WNDPROC** la **DefWindowProc**. Et une autre qui utilise la classe de fenêtre prédéfinie "**Button**". Où sont passés les messages ? Et la fin du programme ?

E. La réponse à la question précédente c'est que les **WNDPROC** copiées soit de la classe de fenêtre variable locale qui utilise **DefWindProc** soit de la classe prédéfinie "Button" ne sont pas les nôtres donc on ne les contrôle pas ! Pour l'instant on ne contrôle que les fenêtres qui utilisent **DefClsWind** ou **LocClsWind**.

Le deuxième problème est que l'on souhaite avoir une méthode **WndMeth** à la place de la méthode statique **WndProc** pour traiter les messages (pour avoir accès aux attributs de la fenêtre et pour mettre en place le polymorphisme d'héritage) mais le passage par **WndProc** est imposé par les API Windows. Il faut donc retrouver l'adresse de l'objet **CWind** alors que l'on se trouve dans la méthode statique mais heureusement on dispose (en argument) du *handle* de la fenêtre.

Les 2 problèmes trouvent les solutions dans les champs **WNDPROC** et **USERDATA** de la structure de notre fenêtre (gardée par le système).

Le champ **WNDPROC** va subir la technique de *sous-classement* après la création de la fenêtre : on sauvegarde la **WNDPROC** obtenue de la classe prédéfinie dans un attribut **defproc** puis on la remplace avec la nôtre à l'aide de **Get/SetWindowLongPtr**. C'est le cas des 2 premiers constructeurs quand ils n'utilisent pas **DefClsWind**. Pour les autres cas on initialise **defproc** avec **DefWindowProc**.

Le champ **USERDATA** va être utilisé pour garder l'adresse de notre objet. Cela se fait à l'aide de **SetWindowLongPtr** en même temps que le *sous-classement*. Mais cela nous fait perdre les premiers messages envoyés pendant la création. Comme on ne veut pas les intercepter pendant la boucle de messages (on a vu que tous les messages ne passent pas par ici) ces premiers messages seront perdus pour les classes prédéfinies. Mais pour les autres voici la solution :

Au début de chaque constructeur on copie systématiquement l'adresse de notre objet dans un attribut statique **copy_this**. Dans **WndProc** on récupère l'adresse de notre objet du champ **USERDATA** à l'aide du *handle* et **GetWindowLongPtr**. S'il est nul et que **copy_this** n'est pas nul alors on l'écrit dans le champ **USERDATA**. Dans le cas improbable que les 2 soient nuls on appelle la **DefWindowProc**, dans le cas contraire on a l'adresse de notre objet et on peut appeler une méthode **WndMeth** avec les mêmes paramètres. Cette méthode reprend le code pour **WM_DESTROY** et les autres messages sont envoyés par défaut vers **defproc**. En réalité la **defproc** pointe vers la **DefWindowProc** si l'on a utilisée une classe de fenêtre à nous ou vers la **WNDPROC** héritée de la classe de fenêtre "étrangère" à notre objet. Dans tous les cas l'enchainement du traitement se fait parfaitement bien. Utiliser les mêmes tests du point **D**.

F. Tester la classe en créant 3 fenêtres principales de taille 200x200. Qui provoque la fin de la boucle de messages et à quel moment sont fermées les autres fenêtres ? Utiliser l'API **MessageBox** pour bien observer.

Cet exemple montre que la fermeture de n'importe quelle fenêtre met fin à la boucle de messages, or cela n'est pas assez souple. On souhaite qu'il y ait un attribut **is_main** dans chaque objet qui détermine si la fermeture de la fenêtre provoque ou non l'arrêt de la boucle de messages. Initialiser-le à **false** pour le 2ème constructeur et à **true** pour les autres. Prévoir un setter **SetMain**.

Tester la nouvelle version améliorée, en désignant comme fenêtre principale que la première.

G. Enrichir **WndMeth** en traitant explicitement (en plus de **WM_DESTROY**) les messages : **WM_PAINT**, **WM_TIMER**, **WM_CLOSE**, **WM_COMMAND**, **WM_CREATE**, **WM_KEYUP**, **WM_KEYDOWN**, **WM_LBUTTONDOWN**, **WM_LBUTTONUP**, **WM_RBUTTONDOWN**, **WM_RBUTTONUP**, **WM_MOUSEMOVE, WM_DROPFILES**. Pour chaque message appeler une méthode virtuelle dont le nom ressemble au message (**WmPaint** etc.) et qui retourne un **LRESULT** nul si elle traite le message, dans le cas contraire on l'envoie (comme avant) vers **DefWindowProc**. Définir des méthodes virtuelles qui (pour l'instant) ne traitent pas les messages (sauf un message de test vers la console). Tester.

Exercice 2 - Utiliser et améliorer les classes

A. Créer 3 fenêtres de taille 200x200 positionnées à 100, 350 et 600 sur X et 200 sur Y, de couleur bleue, blanche et rouge. Seulement la 1ère fenêtre doit fermer l'application.

B. On souhait rajouter une 4ème et 5ème fenêtre de couleur violette et jaune sans aucune zone non-cliente (ni bordure, ni barre de titre). Est-ce que la classe **CWind** permet cela ? Rajouter des méthodes permettant de lire et de modifier le style et le style étendu d'une fenêtre. Attention, après la modification d'un style qui change la taille des zones cliente ou non-cliente il faut appeler **SetWindowPos** avec les flags **NOMOVE**, **NOSIZE**, **NOZORDER**, **NOACTIVATE** et **FRAMECHANGED** pour que la fenêtre soit redessinée correctement. Utilisez les nouvelles méthodes pour produire le bon résultat. Comment faire si l'on veut bouger ces fenêtres ?

C. On souhaite que la première fenêtre demande à l'utilisateur avant sa fermeture s'il veut vraiment quitter l'application. S'il dit non, la fenêtre ne se ferme pas.

D. On souhaite avoir une fenêtre qui affiche dans sa barre de titre (**SetWindowText**) le nombre de secondes depuis qu'elle a démarré. Créer un *timer* qui envoie **WM_TIMER** à l'aide de **SetTimer**.

E. On souhaite avoir une fenêtre qui "colle" à la souris dès que le curseur passe par sa zone cliente (en réalité le centre de la fenêtre doit coller au curseur de la souris). La fenêtre arrête de coller à la souris dès que l'on appuie sur un des boutons. Attention aux différents systèmes de coordonnées !

Exercice 3 - GDI, GDI+ et ressources

A. Créer une classe **CPaintCercles** dérivée de **CWind** qui dessine d'une manière persistante des cercles de fond rouge, de rayon 40 et de centres (50+k*100, 50+i*100) sur toute sa zone cliente. Tester.

B. Enrichir la classe précédente pour que les clicks de la souris provoquent l'affichage non-persistant, à l'endroit du click, des coordonnées du click sous la forme "(x, y)". Utiliser **TextOut** en affichant du texte vert pour les clicks gauches et bleu pour les clicks droits. Tester la persistance de l'affichage.

C. En s'inspirant des fenêtres sans zone NC de l'exercice précédent, concevoir une classe **CFormWind** qui affiche des fenêtres d'une couleur et d'une forme particulière. Son constructeur accepte le rectangle englobant et la couleur du fond. Elle dispose des méthodes pour changer sa forme en ellipse (**SetRound**), en triangle isocèle (**SetTriangle**), en rectangle à coins arrondies (**SetRoundRect**).

Pour réaliser une forme particulière, on utilise l'API **SetWindowRgn** en lui passant la région voulue. Celle-ci peut être "fabriquée" avec **CreateEllipticRgn**, **CreatePolygonRgn**, ou **CreateRoundRectRgn**.

Tester la classe en créant trois fenêtres de 3 formes différentes et de couleur rouge, jaune et verte. Peut-on déplacer ou fermer ces fenêtres ?

Pour assurer leur déplacement à la souris l'on doit faire croire à l'OS que appui sur le bouton gauche tombe sur une zone non-cliente; pour cela il suffit d'intercepter cet appui (**WM_LBUTTONDOWN**) et renvoyer (**SendMessage**) vers la même fenêtre le message **WM_NCLBUTTONDOWN** avec le **WPARAM** égal à **HTCAPTION** (=2) et **LPARAM** nul.

D. Concevoir une classe qui affiche une fenêtre à fond noir et dessine (d'une manière non-persistante) la trace laissée par les mouvements de la souris. La couleur et l'épaisseur de la trace

seront des attributs de valeurs initiales "blanc" et 3. Prévoir de méthodes pour changer ces attributs. Tester.

Modifier la classe pour qu'elle n'affiche que les traces où le bouton gauche est appuyé. Faire en sorte que la fenêtre s'affiche en plein écran, sans de zone NC. Utiliser la touche **BACKSPACE** pour effacer l'écran et d'autres touches pour changer de couleur et / ou d'épaisseur de stylo.

E. Modifier le 1er constructeur de **CWind** pour qu'il puisse accepter comme dernier paramètre un **WORD** signifiant l'**ID** de ressource de type menu (une valeur nulle signifie toujours que l'on ne veut pas de menu). Créer ensuite un nouveau menu constitué d'un *popup* Fichier et un élément Quitter. Tester l'affichage du menu, ensuite trouver une solution pour fermer la fenêtre et l'application si l'utilisateur choisi d'appuyer sur Quitter. Utiliser le fichier "resource.h" crée par l'éditeur de ressources.

F. Concevoir une classe **CTranspWind** qui affiche une fenêtre transparente à 50% de taille 200x200. A chaque click droit, elle augmente de 10% sa transparence et à chaque clique gauche elle la diminue. Conseil : Utiliser le style étendu **WS_EX_LAYERED** puis appeler dans le constructeur l'API **SetLayeredWindowAttributes** avec les bons paramètres. (cette API n'est accessible que pour une version Windows **_WIN32_WINNT** >= 0x0500) .

G. Concevoir une classe **CHoleWind** qui affiche une fenêtre avec un trou au milieu dont le diamètre fait toujours 80% de la taille de la fenêtre. Même s'il est possible d'utiliser l'approche du point **C.**, utiliser plutôt le style étendu **WS_EX_LAYERED** puis appeler dans le constructeur l'API **SetLayeredWindowAttributes** avec les bons paramètres. Prévoir une méthode privée qui dessine un cercle de taille et couleur adaptée (à l'aide de l'API **Ellipse**) en utilisant le contexte graphique fourni en paramètre. Appeler-la du constructeur avec un DC obtenu par **GetDC** et du **WmPaint** avec le DC obtenu par **BeginPaint**. *Remarque* : le contenu des fenêtres transparentes est exceptionnellement mémorisé et affiché par l'OS. Si la fenêtre ne change pas de taille, il n'est pas nécessaire de la redessiner en réponse à **WM_PAINT**; un seul dessin suffit (à la manière non-persistante, en utilisant **GetDC**) et l'OS va afficher le contenu d'une manière persistante jusqu'au prochain changement (dessin).

H. Reprendre le point **A.** de l'exercice **2** en rajoutant 2 fenêtres de même taille dont le nom de classe de fenêtre est "**Edit**" et "**Button**". Peut-on intercepter les messages qui arrivent pour les 2 fenêtres ? Pourquoi ? Que proposez-vous ?
Comment aligner à droite ou à gauche le texte saisi dans la fenêtre d'édition ?

I. Concevoir une classe **CEditCopy** affichant une fenêtre de taille fixe (fond gris) qui possède 4 enfants : 2 boutons et 2 fenêtres d'édition. La création des enfants doit se faire dynamiquement dans le constructeur de la classe et il faut garder leur adresse comme attribut pointeur (attention à la gestion des ressources). Donner des **ID** différents pour les 4 enfants. Intercepter les messages **WM_COMMAND** émises par les 2 boutons (où se trouve leur **ID** ?) pour déclencher une copie du contenu d'une fenêtre d'édition vers l'autre et inversement. Toute la gestion se fait dans la classe. Utiliser l'envoi des messages **WM_SETTEXT** et **WM_GETTEXT** pour la copie.

Rajouter un menu à cette classe (*popup* **Fichier** et **Edition**) proposant de quitter ou de copier d'une fenêtre d'édition vers l'autre. Utiliser intelligemment les ID des menus et des boutons.

J. Concevoir une classe **WinGDI** dérivé de **CWind** qui initialise la bibliothèque GDI+ et propose une méthode virtuelle **PaintGDI** appelée à chaque message **WM_PAINT**, qui offre à l'utilisateur dans l'argument d'entrée la référence du contexte graphique GDI+ **Graphics** de la fenêtre courante.

Tester cette méthode en affichant à l'écran quelques segments de couleurs différentes, du texte et/ou des formes géométriques.

K. Dériver de **WinGDI** une classe **WinJO** qui affiche en permanence l'emblème des jeux olympiques adaptées à la taille de la fenêtre (respecter les indications fournies sur les transparents du cours).

L. Créer une classe **WinImg** dérivée de **WinGDI** qui affiche une image dont le nom du fichier est fourni à la construction et la taille est fixe (celle de l'image) ou ajustée à la fenêtre, en fonction d'un booléen. Tester (ajouter un menu ou un événement qui change le booléen pendant l'exécution).

Enrichir la classe **WinImg** pour que l'on puisse tirer (glisser) un fichier image à l'intérieur de la fenêtre (pour activer ce comportement utiliser l'API **DragAcceptFiles**). Si elle existe bien, cette image doit remplacer l'image courante. ***Attention à la bonne gestion de la mémoire !***

Surimposer sur l'image les informations la concernant (nom du fichier, nombre de pixels etc.)

Attention : la GDI+ n'est disponible qu'en version Unicode. Tester donc votre classe en mode Unicode pour valider toutes les fonctionnalités puis adapter (enrichir) le code pour que les mêmes fonctionnalités puissent être disponibles en mode ANSI/multibyte.

Exercice 4 - *Multithreads*, multiprocessus, synchronisation et communication

Créer une fonction principale **WinMain** (ANSI/Unicode) qui exécute différentes fonctions de test (**test1**, **test2**, **test3** ...) en fonction du premier caractère de la ligne de commande (ou **test0** si ligne de taille nulle ou caractère non-prévu).

A. Créer une classe **CWorkWind** qui affiche une fenêtre de taille fixe avec un menu contenant les éléments **Quitter**, **WorkSync**, **WorkAsync**.

Prévoir une méthode **SlowWork** qui doit simuler un travail long (quelques secondes). A l'aide d'un contexte graphique non-persistant de la zone client on affiche des lignes horizontales d'une même couleur (choisie aléatoirement), du haut en bas de la fenêtre (de y=0 à 200). Pour ralentir plus, entre 2 lignes on se met 50 ms en sommeil à l'aide de **Sleep**. Appeler la méthode en cliquant sur **WorkSync** et tester le résultat : vitesse, réactivité de l'application, réaction du système ... Quelle est l'explication ?

B. Ecrire une méthode statique **WorkAsyncTh** qui sera lancée comme *thread* par l'appui de **WorkAsync** (attention au prototype) en lui passant comme argument l'adresse **this** de l'objet. La méthode statique récupère l'argument, le typecaste vers **CWorkWind*** pour appeler la méthode **SlowWork**. Tester et comparer les résultats par rapport au lancement synchrone. Peut-on lancer plusieurs *threads* en parallèle ? Utiliser le gestionnaire de tâches (en actualisation rapide) pour visualiser le nombre de *threads* du processus.

C. Synchroniser l'accès à **SlowWork** à l'aide d'un *mutex* créé dans le constructeur (attention à la gestion de cette ressource). Choisir un *timeout* d'environ 10s (élément de menu **WorkSyncMutex**, *thread* **WorkSyncMutexTh** et méthode **WorkSyncMutex**). Tester le nouveau comportement en lançant plusieurs threads ... Vérifiez leur nombre avec le gestionnaire de tâches.

D. Réaliser une version légèrement différente (activée par le menu **WorkSyncSectC**) où la synchronisation par *mutex* est remplacée par une section critique. Tester et noter les éventuelles différences.

E. Synchroniser l'accès à **SlowWork** à l'aide d'un sémaphore de valeur maximale 3 (élément de menu **WorkSyncSema**, *thread* **WorkSyncSemaTh** et méthode **WorkSyncSema**). Tester et visualiser les *threads* en action. Afficher à la console la valeur du sémaphore avant chaque **WaitForSingleObjet** (comment peut-on lire sa valeur ?).

F. On souhaite réaliser une synchronisation par événement entre plusieurs processus. Un processus jouera le rôle du manager (il va créer et activer d'une manière interactive l'événement "**WorkEvt**"). D'autres processus (3 par exemple) vont proposer à l'utilisateur de déclencher un *thread* similaire au point C (appel de **SlowWork**), mais la synchronisation sera faite par l'événement créé et signalé par le manager.

Pour cela on conçoit une classe **CManagWind** de taille fixe et avec menu (sur le modèle de la **CWorkWind**). Elle aura un élément de menu **Start** qui va activer pour une courte durée de temps (100 ms) l'événement préalablement créé dans le constructeur (attention à la gestion de la ressource). Cette classe sera instanciée dans une fonction de test qui sera appelée par défaut par **WinMain**.

En même temps on modifie la classe **CWorkWind** en lui rajoutant comme attribut un événement ouvert dès la construction (le même evenement "**WorkEvt**" que celui créé par **CManagWind** !). En appuyant sur **WorkSyncEvt** l'utilisateur à la possibilité de lancer un *thread* **WorkSyncEvtTh** qui appelle la méthode **WorkSyncEvt.** Celle-ci se synchronise sur l'événement préalablement ouvert et appelle ensuite **SlowWork**. Utiliser un *timeout* assez long (10s). Cette classe sera

instanciée dans une fonction de test qui sera appelée par **WinMain** si le 1er caractère de la ligne de commande est '1'.

Rajouter à **CManagWind** une méthode **LaunchProcess** activée par un élément de menu de même nom qui lance avec **CreateProcess** le même processus avec la ligne de commande "1" (par exemple "exo4 1").

Test : Lancer le processus sans arguments pour avoir accès à la fenêtre **CManagWind** puis lancer 2-3 processus en sélectionnant **LauchProcess**, qui va faire apparaître des fenêtres de type **CWorkWind**. Pour chaque fenêtre, sélectionner **WorkSyncEvt** pour enclencher le *working thread* synchronisé sur l'événement créé par le 1er processus. Ensuite activer l'événement dans **CManagWind** en appuyant sur **Start** et regarder le résultat produit. Regarder les informations affichées par le gestionnaire de tâches.

F. Modifier les classes du point précédent pour réaliser non pas un seul appel mais un nombre infini d'appels de **SlowWork**, synchronisés par un seul événement. Cela suppose la copie et la modification de la méthode **WorkSyncEvt** dans une méthode **WorkSyncEvtLoop** qui dans une boucle infinie se synchronise sur l'événement en question puis appelle **SlowWork**. La sortie de la boucle se fait par le dépassement du timeout (5-10 secondes).