Interfaçage de WinDev avec un SDK en DLL natives

Sommaire

1. Comment se présente un SDK

Il peut y avoir plusieurs types de SDK : DLL, Active-X (objets OLE Automation), assemblages .Net. Les DLL sont moins pratiques, car une DLL ne contient aucune déclaration.

Donc, une DLL ne permet pas de savoir :

- Combien de paramètres sont attendus par telle fonction
- A quoi correspond le code d'erreur 42
- Avec quelle convention d'appel une fonction doit être appelée (c'est-à-dire : comment traduire un appel de fonction en langage machine, avec le passage de paramètres etc.)

Malgré ces problèmes, ce type de SDK est le plus léger/performant, le plus compatible et le plus usité.

1.1 Headers C

En C, un header est un fichier de code source normal, mais on le nomme « header » car il contient du code qu'on voudrait insérer au début de plusieurs autres fichiers.

Il a pour extension « .h » et il est inséré dans un autre code source avec l'instruction :

```
#include "toto.h"   // Insère tout le contenu de toto.h ici
```

Le header contiendra du code de déclaration :

- Je déclare qu'une fonction nommée « `Bobby` » existe, qu'elle prend en paramètre 2 entiers et qu'elle retourne une chaîne.
- Je déclare que la constante nommée « `John` » est un entier valant 42.
- Je déclare que le type « `Billy` » est une structure composée de 2 entiers.

Grâce à ce code de déclaration, un code d'implémentation[1] pourra appeler la fonction `Bobby` sans qu'elle soit présente dans le même code source, utiliser la constante `John` que le compilateur remplacera par 42 et manipuler des variables de type `Billy`.

Utiliser un SDK en C/C++ revient donc à insérer une ligne `#include` et ensuite utiliser les fonctions de la DLL comme si on les avait écrites nous-mêmes.

Dans WinDev, c'est beaucoup moins facile car il faut compenser l'absence de l'instruction `#include`.

[1] Code d'implémentation : code qui implémente le contenu des fonctions, à opposer au code de déclaration qui ne fait que déclarer que les fonctions vont exister, et donc qu'on peut y faire référence dans du code.

1.2 Documentation

La documentation d'un SDK sera écrite en considérant que le lecteur comprend le C ou un langage proche.

Il faut prêter attention aux choses suivantes :

- Les types de données attendus. Un entier peut être sur 8, 16, 32 ou 64 bits, un réel sur 32 ou 64, une chaîne peut être un pointeur ou un tableau, en 8, 16 ou 32 bits, en UTF ou UCS etc. Grâce aux entiers « little-endian[2] » des processeurs Intel, une erreur sur la taille d'un entier pourra parfois passer inaperçue, mais on risque alors de rencontrer des effets de bord plus tard.

- Les limites de chaque fonction, notamment quelles valeurs de paramètres sont interdites.

- Les codes d'erreurs : quels moyens avons-nous de diagnostiquer un échec ou l'utilisation inappropriée d'une fonction ?

- La notion de propriété des données :

    - Une DLL peut fournir une fonction de création de données, ou création d'un objet, puis des fonctions de manipulation qui prendront en paramètre un pointeur ou un handle (pointeur indirect) vers l'objet. Dans ce cas il faudra penser à utiliser la DLL également pour libérer cette ressource.

    - Mais elle peut aussi exiger que le programme client alloue de la mémoire qu'elle va ensuite manipuler. En toute logique, ça sera alors au programme client de gérer cette mémoire, et de l'allouer et la libérer aux bons moments et en utilisant les moyens du bord.

    - Enfin, il peut y avoir des cas vicieux, comme par exemple une DLL qui utilise un buffer statique[3] pour échanger des données. Une fonction renvoie une chaîne : en réalité, elle renvoie un pointeur vers un buffer statique dans lequel elle a écrit la chaîne. Au prochain appel, si le programme n'a pas encore dupliqué la chaîne, elle sera écrasée.

---

[2] Little-endian : consiste à inverser les octets d'un entier, de manière à ce que le poids faible soit au début : l'entier 5142 vaut 1416 en hexadécimal. Il est composé des octets : `00 00 14 16`. Mais en mémoire il sera sous la forme : `16 14 00 00`. Si on accède à cet entier en considérant qu'il est sur 2 octets au lieu de 4, on lira `16 14` = 5142, soit la même valeur, au lieu de `00 00` qui aurait pu provoquer un bug.

[3] Buffer statique : zone de mémoire allouée au chargement de l'exécutable, jamais déplacée, jamais détruite.

2. Comment lire un header C

Un header C devrait normalement être utilisé tel quel, et sert à préparer le terrain pour l'utilisateur, en prenant en charge toutes les complications liées aux problèmes de portabilité etc.

De fait, il peut être compliqué à détricoter pour le porter dans un autre langage.

2.1 Préprocesseur

Le langage C est en fait composé de 2 langages :

- Le code C compilable.
- Le langage du préprocesseur, ou langage de « macro ».

Le langage de préprocesseur est entremêlé avec le code, un peu comme quand on mélange du HTML et du PHP ou du JavaScript. Il est interprété en premier afin de générer un code source définitif qui sera passé au compilateur.

Les instructions du préprocesseur sont différenciées grâce au caractère « # » au début de chaque ligne.
C'est un langage qui est totalement indépendant du C et qui permet notamment de faire des macros, c'est-à-dire créer des mots clés derrière lesquels se cache du code, ou bien faire du code cible conditionnel[4], c'est-à-dire gérer les spécificités des plateformes cibles[5] (OS notamment).

Par exemple :

```
#define Bob if (i == 1) return 2;
```

Si j'écris ensuite du code contenant le mot `Bob`, le préprocesseur le remplacera par `if (i == 1) return 2;`, et c'est ce que le compilateur C verra au final.

Donc, le préprocesseur permet de créer un nouveau vocabulaire, et pour comprendre le header d'un SDK, il faudra parfois repérer ce changement de vocabulaire.

Exemple :

[4] Code cible conditionnel : « si je compile l'exe pour Windows 32 bits avec Visual Studio, alors utiliser tel code source ». La différence avec un programme dont la logique s'adapte à la plateforme d'exécution (`SI EnMode64bits() ALORS…`), c'est que le code est adapté avant même la compilation, et chaque exécutable ne contiendra que le code ciblé.

[5] Plateforme cible : environnement qui va accueillir le programme final : système d'exploitation, matériel, et même drivers et frameworks.

```
#ifdef WIN64        // Si le programme est compilé pour Windows 64 bits
#define F F_x64     // Appeler F() reviendra à appeler F_x64()
#define MAXSIZE 400 // La constante MAXSIZE vaudra 400
#else               // Sinon, si on compile pour du 32 bits
#define F F_x32     // Appeler F() reviendra à appeler F_x32()
#define MAXSIZE 200 // La constant MAXSIZE vaudra 200
#endif
```

Ce code signifie que grâce au préprocesseur, un utilisateur du SDK pourra appeler la fonction `F()` et utiliser la valeur `MAXSIZE` sans ce soucier de la plateforme pour laquelle il développe. Le code sera automatiquement adapté au 32 ou au 64 bits.

Mais ce qui est réellement présent dans la DLL c'est 2 fonctions : `F_x32` et `F_x64`. Quant à `MAXSIZE`, ça n'existe même pas dans la DLL, ce n'est qu'un nombre à insérer dans le code client.

Comme WinDev n'utilise pas ce header, il n'a pas ce vocabulaire, et donc le développeur doit appeler explicitement `F_x32()` ou `F_x64()`, et déclarer lui-même la constante `MAXSIZE` qui va bien.

Le problème est que la documentation du SDK parlera uniquement de la fonction `F()`.

Un autre usage du préprocesseur est les extensions de langage, c'est-à-dire des fonctionnalités ajoutées au C/C++, pour un compilateur ou une plateforme spécifique. Ces fonctions sont accessibles par le mot clé `#pragma`.

Elles sont rarement utilisées dans un header de SDK, car non portables, mais il y en a une à laquelle il faut faire attention :

```
#pragma pack(4)
```

Le pragma pack indique l'alignement des membres de structures. Il s'agit d'alignement en mémoire, c'est-à-dire qu'un membre devra être positionné à une adresse mémoire multiple de 4 dans l'exemple précédent. La conséquence est qu'une structure peut contenir du vide entre certains membres. Comme c'est un sujet un peu complexe, on en reparlera dans un prochain paragraphe.

2.2 Définition de types

En C, on peut définir n'importe quel type de donnée : classe, structure, mais aussi scalaires[6], tableaux et pointeurs.

Ça veut dire que si je veux renommer le type `int` en `tataouine`, je peux :

[6] Type scalaire : type de données représentant un nombre, et plus particulièrement un type natif que le microprocesseur sait manipuler.

```
typedef int tataouine;     // Définition du type tataouine

tataouine Fonction()       // Fonction retournant un tataouine
{
    tataouine i = 1;       // Déclare un tataouine initialisé à 1
    tataouine j = 2;       // Déclare un autre tataouine égal à 2
    return i + j;          // Renvoie un tataouine égal à 3
}
```

Ainsi, une infinité de mots clés peuvent désigner le même type de donnée.

Et c'est couramment utilisé, même pour les types scalaires, car les types standard ont souvent un nom mal adapté : `unsigned int` est long à écrire et ne précise pas la taille exacte de la variable, on lui préfère donc `DWORD` (API Windows) ou `uint32_t` (standard C/C++ récent).

2.2.1 Types scalaires

Les types scalaires sont les types qui peuvent seulement contenir un nombre. C'est le seul type réellement géré par le microprocesseur, tous les autres sont en réalité construits par agrégation de scalaires.

Les scalaires du C sont les suivants :

- `int` : Entier 32 bits signé. Valeurs possibles : [-2147483648, 2147483647]
- `long` : Synonyme de `int`. Historiquement, un `int` pouvait être sur 16 bits, d'où le `long`.
- `unsigned int` : Entier 32 bits non signé. [0, 4294967295]
- `unsigned` : Synonyme de `unsigned int`
- `short` : Entier 16 bits signé. [-32768, 32767]
- `unsigned short` : Entier 16 bits non signé. [0, 65535]
- `char` : Entier 8 bits signé. [-128, 127]
- `unsigned char` : Entier 8 bits non signé. [0, 255]
- `long long` : Entier 64 bits signé. [-9223372036854775808, 9223372036854775807]
- `unsigned long long` : Entier 64 bits non signé. [0, 18446744073709551615]
- `float` : Nombre à virgule flottante base 2, sur 32 bits.
- `double` : Nombre à virgule flottante base 2, sur 64 bits.

2.2.2 Valeurs immédiates

Une valeur immédiate est une valeur écrite en dur dans le code, par opposition à une variable, même constante.

Voici toutes les formes de valeurs immédiates en C :

```
42     // Valeur d'un entier
42u    // Valeur d'un entier avec suffixe pour préciser son type
       // (ici c'est u pour unsigned)
0x2A   // Valeur hexadécimale, 0x2A = 42
052    // Valeur octale (préfixe 0), 052 = 42
42.0   // Valeur à virgule flottante (réel)
42.0f  // Valeur à virgule flottante avec suffixe (réel 32 bits)
'A'    // Caractère, 'A' = 65 (code ASCII)
'\x41' // Caractère par escaping : '\x41' = 0x41 = 65 = 'A'
'\65'  // Caractère par escaping : '\65' = 65 = 'A'
"ABC"  // Chaîne de caractères (on verra plus tard que celle-ci en a 4)
L"ABC" // Chaîne de caractères Unicode (UCS-2 ou UTF-16 sous Windows)
```

Il y a 2 manières de nommer des valeurs immédiates en C :

- `#define` permet de donner un nom à n'importe quel bout de code, donc il permet aussi de nommer n'importe quelle valeur immédiate.
- `enum` permet de définir une suite de nombres entiers.

Concernant `enum`, voici un exemple :

```
enum CodeErreur
{
    ERR_OK,                // Par défaut, le 1er membre d'un enum vaut zéro
    ERR_INVALID_PARAM,     // Ce code vaut 1
    ERR_NOT_ENOUGH_MEM,    // Celui-ci vaut 2
    ERR_EARTH_QUAKE = 42,  // Mais on peut aussi forcer une valeur
    ERR_MELTDOWN,          // Cette valeur vaut 42 + 1 = 43
    ERR_GOOD = 0           // On peut avoir 2 noms pour une même valeur
};
```

Quand une fonction prend en paramètre un type `enum`, ça permet de savoir quelles valeurs sont autorisées, on ne va jamais passer 51 si ça ne fait pas partie de l'énumération.

Quand une fonction renvoie un type `enum`, ça permet de savoir exactement quelles valeurs on risque de récupérer, et de faire une gestion d'erreur exhaustive.

On peut parfois rencontrer des expressions constantes en guise de valeurs immédiates, c'est notamment le cas pour des masques de bits[7] :

```
enum Flags
{
    GotCheese       = 1 << 0,           // = 1
    GotMilk         = 1 << 1,           // = 2
    GotBread        = 1 << 2,           // = 4
    GotSalt         = 1 << 3,           // = 8
    GotAnything     = (1 << 4) - 1,     // = 15 (1 + 2 + 4 + 8)
    OtherFlags      = ~((1 << 4) - 1)   // = 0xFFFFFFF0 (inversion de 15)
};
```

Donc il est préférable de connaître les opérateurs de base pour pouvoir recopier ces nombres dans WinDev :

- << : décalage de bits vers la gauche (WD : bitDécaleGauche)
- >> : décalage de bits vers la droite (WD : bitDécaleDroite)
- ~ : inversion des bits (WD : NONBinaire ou ~)
- ^ : XOR (ou exclusif) bit à bit (WD : OUExclusifBinaire ou ||)
- & : AND bit à bit (WD : ETBinaire ou &)
- | : OR bit à bit (WD : OUBinaire ou |)

2.2.3 Pointeurs

Un pointeur est un entier utilisé pour stocker une adresse mémoire.

Sa taille est donc définie par le système d'adressage de la plateforme d'exécution. Sur un système 32 bits, un pointeur est donc équivalent à un `unsigned int`.

La déclaration d'un pointeur se fait en préfixant la variable par une étoile :

[7] Masque de bits : entier dont la valeur est choisie pour la position de ses bits. Par exemple, 4 vaut 0100 en binaire, soit un seul bit actif. 2 vaut 0010. Si on dit par exemple que 4 veut dire « lecture » et 2 veut dire « écriture », on peut écrire une fonction prenant en paramètre 0, 2, 4 ou 4 + 2, pour dire « rien » (0000), « écriture » (0010), « lecture » (0100), « lecture + écriture » (0110). C'est comme un mini-tableau de booléens, et c'est ce que WinDev appelle des « constantes combinables », par exemple sur la fonction fOuvre.

```
int *ptr1;    // Adresse d'un int
void *ptr2;   // Adresse sans type
char **ptr3; // Adresse d'un pointeur qui est lui-même l'adresse d'un char
short* ptr4; // Certains développeurs collent l'étoile au type...
short * ptr5;// d'autres la mettent au milieu, mais le résultat est le même

// Code vicieux :
// - Déclaration de 2 int : i et j
// - Et déclaration d'une adresse : ptr6
// Le tout sur la même ligne
int i, *ptr6, j;
```

Une définition de type permet de ne pas avoir à écrire l'étoile partout :

```
typedef int *IntPtr;       // Définition du type IntPtr
typedef int **IntPtrPtr;   // Définition du type IntPtrPtr

IntPtr ptr1; // Adresse d'un int
IntPtr *ptr2;// Adresse d'un pointeur qui est lui-même l'adresse d'un int
IntPtrPtr ptr3;     // Même chose que ptr2
```

Quand un pointeur ne pointe nulle part, par exemple après la destruction de l'objet sur lequel il pointait, la convention est de le mettre à zéro, car ça permet de savoir qu'il est invalide.

Dans l'API Windows, la constante `NULL` vaut zéro et elle est utilisée pour dire « pas de données » quand on l'affecte à un pointeur ou un handle[8].

L'utilisation de pointeurs est très sensible : il n'existe pas de moyen réellement fiable et stable de tester la validité d'un pointeur. Passer une adresse invalide à une fonction de SDK peut avoir pour conséquence de lui faire écraser des données ou du code en mémoire, de rendre une application instable et de la faire planter.

Certaines fonctions de SDK accepteront un pointeur nul en paramètre pour rendre le paramètre optionnel. Mais si ce n'est pas le cas, passer un pointeur nul fera simplement planter la fonction.

2.2.4 Tableaux C

En C, un tableau ne se manipule que par l'adresse du premier élément. En mémoire, il n'y a que le contenu du tableau.

[8] Handle : numéro pointant sur un objet, de la même manière qu'une adresse, mais destiné à être interprété par un gestionnaire. Cette indirection supplémentaire permet plus de sécurité, car le gestionnaire va vérifier que le handle existe bien dans sa liste pour y trouver l'adresse réelle de l'objet, adresse qu'il est seul à manipuler. L'API Windows utilise ce système.

```
int tableau[42];    // Tableau de 42 entiers
int *ptr;

ptr = tableau;      // Ici, on copie une adresse seulement
ptr[1] = 2;         // Affectation du 2ème élément
tableau[0] = 1;     // Affectation du 1er élément
// A ce stade, tableau[] commence par { 1, 2 }
*ptr = 3;
*(tableau + 1) = 4;
// Maintenant, tableau[] commence par { 3, 4 }
```

Donc, un pointeur et un tableau ne sont pas distinguables en C. Parfois, une fonction prenant en paramètre un pointeur attendra un seul élément, parfois elle en attendra plusieurs. Seule la documentation du SDK peut permettre de savoir ce qui est attendu.

2.2.5 Chaînes C

En C++, Java, WinDev, une chaîne se manipule aussi facilement qu'un scalaire, mais en interne c'est un objet composé d'un entier et d'un pointeur, et chaque opération comme le « + » ou le « = » est en réalité un appel à une méthode de l'objet.

En C, une chaîne est réduite à sa forme la plus simple : une suite de caractères en mémoire dont on ne connaît que l'adresse de départ. C'est un tableau C.

```
char *chaine = "Hello world";    // Adresse d'une série de caractères
```

Un caractère n'est qu'un entier sur 8, 16 ou 32 bits. Le plus commun étant le 8 bits (`char`) pour l'instant, les autres tailles servant à implémenter l'Unicode (`wchar_t`).

Comme on n'a que l'adresse du premier caractère, on utilise un caractère terminateur pour savoir où est la fin de la chaîne. Ainsi, `"Hello world"` est une chaîne de longueur 11, mais qui contient en réalité 12 caractères, le dernier étant le caractère nul (valeur 0).

```
char *chaine = "Hello world";
chaine[0] = 'h';
chaine[5] = 0;
// A ce stade, la chaîne est "hello"
chaine[5] = '-';
// Maintenant c'est "hello-world"
chaine[4] = '\0';   // '\0' == 0
// Maintenant c'est "hell"
```

Attention : Il est tout à fait possible qu'un SDK choisisse une autre manière de gérer la longueur d'une chaîne. Par exemple, on peut avoir une seconde variable qui contient cette taille, ou bien on peut imposer une longueur maximum et ne pas imposer que le tout dernier caractère soit nul.

Imaginons alors qu'un programme essaie de lire une telle chaîne comme si elle était « null-terminated » :

```
// La chaîne suivante n'a pas de terminateur
// Elle n'est composée que de 4 caractères, tous différents de '\0'
char chaine[4] = { 'A', 'B', 'C', 'D' };
// Calcule la taille de la chaîne en cherchant le caractère nul
int i = 0;
while (chaine[i] != 0)
    ++i;
// La longueur (4) devrait être dans i
// Cependant, la longueur trouvée est aléatoire, et parfois ça plante
```

Le problème de ce code, c'est qu'il va lire les données qui sont après la chaîne en mémoire, ça peut être n'importe quoi. La boucle va donc s'arrêter quand on trouvera un zéro par hasard, ou bien quand on arrivera sur une zone mémoire qui n'est pas attribuée au processus et qui va donc provoquer un plantage.

C'est pourquoi il est important de comprendre comment le SDK va gérer les chaînes.

2.2.6 Types structurés

En C, les types structurés sont de simples agréations, comme les variables composées en WinDev :

```
struct { int x; int y; } vecteur; // Variable composée de 2 entiers

vecteur.x = 1;
vecteur.y = 2;
```

2.2.6.1 Déclaration d'une structure

Dans l'ancienne norme du C, le type `struct` exigeait de répéter le mot `struct` à chaque déclaration de variable.

```
struct TypeVecteur { int x; int y; };   // Déclaration du type

struct TypeVecteur vecteur;      // Déclaration de la variable

vecteur.x = 1;
vecteur.y = 2;
```

Afin de déclarer un « vrai » type, il était donc nécessaire d'utiliser `typedef`.

C’est pourquoi beaucoup de SDK utiliseront cette syntaxe :

```
// C’est le même code que le précédent, mais avec des retours à la ligne
// pour la lisibilité.
// L’autre différence, c’est le typedef qui permet de déclarer un type
// à la place d’une variable.
typedef struct
{
    int x;
    int y;
} TypeVecteur;
// Maintenant que le type existe, déclaration d’une variable
TypeVecteur vecteur;

vecteur.x = 1;
vecteur.y = 2;
```

Parfois, le type « pointeur sur la structure » sera également déclaré dans la foulée :

```
typedef struct
{
    int x;
    int y;
} TypeVecteur, *TypePointeur;

TypeVecteur vecteur;      // Déclare un objet
TypePointeur ptr;         // Déclare un pointeur pour ce type d’objet
```

Plus rare dans les SDK, voici la syntaxe moderne :

```
// En C/C++ moderne, la déclaration d’une structure crée
// un type comme les autres.
struct TypeVecteur
{
    int x;
    int y;
};

typedef TypeVecteur *TypePointeur;

TypeVecteur vecteur;
TypePointeur ptr;
```

2.2.6.2 Structures imbriquées

Evidemment, une structure peut contenir d'autres structures, et même des tableaux :

```
struct MaStruct
{
    int tableau1[4];
    int *tableau2;
    TypeVecteur vecteur;
};
// Le contenu de tableau1 fait réellement partie de la structure.
// Le contenu de tableau2 sera ailleurs dans la mémoire,
// seul un pointeur est présent dans la structure.
// Enfin, le vecteur (x, y) fait partie intégrante de la structure.
//
// Voici une structure parfaitement identique en mémoire :
struct MaStruct2
{
    int tableau1_0;
    int tableau1_1;
    int tableau1_2;
    int tableau1_3;
    int *tableau2;
    int vecteur_x;
    int vecteur_y;
};
```

2.2.6.3 Alignement des structures

Un microprocesseur est optimisé pour manipuler des mots[9] d'une certaine taille.

Sans rentrer dans les détails, quand un processeur accède à la mémoire pour charger un scalaire dans un registre[10], il ne peut le faire que si le scalaire est positionné à une adresse « ronde ».

---

[9] Mot : entier d'une taille supérieure à l'octet, qui reflète généralement la taille des registres du microprocesseur. Un int est un mot sur 32 bits, et on l'appelle souvent « double mot », ou « double word », car il est 2 fois plus grand que le mot 16 bits qui était la norme dans les anciens processeurs. Ainsi, on appellera parfois « quad word » les entiers 64 bits.

[10] Registre : mémoire interne du processeur faisant office de variable pour toutes les opérations qu'il est capable de réaliser. Quand on code « a = b + c », le processeur va charger le contenu de b et de c dans 2 registres, exécuter l'addition, et copier le registre contenant le résultat par-dessus la variable a. Tous nos programmes font des milliards d'opérations sur les registres.

Ça veut dire que si le scalaire est sur 32 bits, soit 4 octets, il pourra le charger depuis l'adresse 1024, ou bien l'adresse 36, mais pas depuis l'adresse 421.

En réalité, il pourra le faire, mais ça risque de ralentir l'exécution.

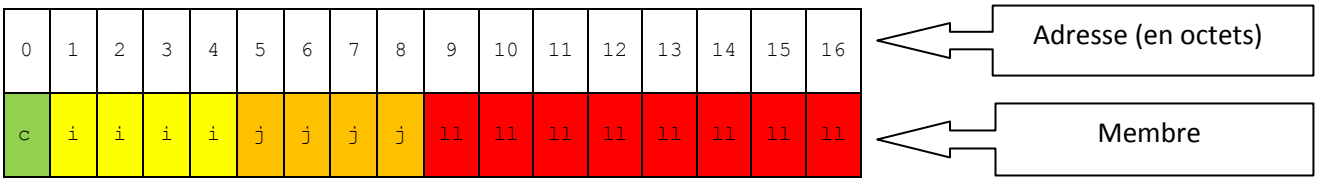
Comme une structure est une série de variables hétérogènes, contrairement aux tableaux qui sont homogènes, le problème de l'alignement ne concerne pas que l'adresse de la structure, mais aussi l'adresse de ses membres :

```
// Cette structure pose un problème d'alignement :
struct MaStruct
{
    char c;         // Entier sur 8 bits (1 octet)
    int i;          // Entier sur 32 bits (4 octets)
    int j;          //
    long long ll;   // Entier sur 64 bits (8 octets)
};
```

La taille théorique de la structure ci-dessus est de 1 + 4 + 4 + 8 = 17 octets.

En réalité, elle prendra 24 octets.

En effet, si une variable de ce type était allouée à l'adresse 0, voici ce qu'on aurait :



Les variables `i` et `j` sont en 1 et 5 qui ne sont pas divisibles par 4.

La variable `ll` est en 9, ce qui n'est pas divisible par 8. Le processeur ne va pas aimer.

Voici donc la disposition qui va être réellement utilisée :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
c				i	i	i	i	j	j	j	j					ll	ll	ll	ll	ll	ll	ll	ll

Les parties blanches ne seront théoriquement pas accessibles depuis le code C, et donc elles sont indéterminées, elles peuvent contenir des zéros, ou bien n'importe quel résidu d'une précédente utilisation de la mémoire.

Pour régler le problème, on aurait pu décider que la structure serait toujours positionnée à une adresse de la forme 8n – 1, puisque `c` ne requiert pas d'alignement spécifique. On aurait ainsi économisé 7 octets par objet. Mais le problème est qu'il faut pouvoir créer des tableaux d'objets, et

les tableaux sont des types compacts n'introduisant pas d'espacement entre 2 éléments. Dès le deuxième élément du tableau, on aurait retrouvé un problème d'alignement.

C'est aussi pour ça qu'on peut trouver des structures avec du vide à la fin. C'est uniquement pour que l'intérieur de la structure reste aligné quand il est répété dans un tableau.

Les règles d'alignement ne sont pas gravées dans le marbre. Leur importance dépend des processeurs, et les compilateurs permettent de les modifier.

Ainsi, l'alignement par défaut actuel est de 8 octets, c'est la taille d'un mot 64 bits. Parfois, un alignement sur 16 ou 32 octets permettra des optimisations. Dans d'autres cas, un alignement sur 4 octets sera nécessaire pour rester compatible avec un ancien code. Et enfin, on voudra parfois privilégier l'espace mémoire à la vitesse en supprimant tout alignement.

Quand on utilise un SDK, on peut s'attendre à un alignement minimal de 4 octets.

Vu que le header doit permettre une utilisation immédiate du SDK, il précisera le niveau d'alignement quand celui par défaut n'est pas le bon. C'est fait avec la commande suivante :

```
#pragma pack(4)
// A partir d'ici, toutes les structures respecteront l'alignement 32 bits

#pragma pack(1)
// Les structures déclarées ici seront aussi compactes que possible
```

Ce pragma est un des seuls à être commun à tous les compilateurs C/C++, et c'est justement à cause de son rôle dans la portabilité d'un code source.

2.2.6.4 Les unions

Les unions sont un type très spécifique au C, pouvant être confondu avec les structures, mais pourtant complètement différent :

```
union MonUnion
{
    int i;
    char texte[4];
};

MonUnion u;

u.i = 0x00323334;
// A présent, la chaîne u.texte contient : "432" !
```

Ce que fait l'union, c'est placer tous ses membres à la même adresse.

Donc en réalité, il n'y a qu'une donnée, et chaque membre de l'union permet de l'interpréter différemment.

Dans l'exemple précédent, on affecte une valeur au membre `i` de type entier, mais si on lit ensuite le membre `texte` de type chaîne (ou tableau de caractère), on y trouvera la valeur affectée à `i`, mais n'ayant plus du tout le même sens ! Il ne s'agit pas d'un type-cast[11].

Notre `union` est composée de 4 octets en mémoire, interprétables au choix comme un entier 32 bits ou un tableau de 4 caractères.

Si une `union` contient des membres de tailles différentes, c'est évidemment la taille du plus grand membre qui sera allouée en mémoire, mais tous seront positionnés au début de la zone mémoire.

Voici un exemple d'utilisation : comment créer un type « Variant » en C, en utilisant aussi peu de mémoire que possible, et notamment sans pointeur ? Le problème d'un Variant est que son adresse doit rester la même quand il change de type, c'est pourquoi il nécessite normalement de déporter ses données et maintenir un pointeur vers celles-ci. Mais on peut s'en passer avec les unions :

```
struct Variant
{
    union    // Données du Variant
    {
        int entier;
        float reel;
        char chaine[10];
        void *pointeur;
    };
    char typeVariant;     // Type du Variant
};

Variant var; // Déclaration d'une variable de type Variant

// On transforme le Variant en entier 'i' et on lui affecte 42
var.typeVariant = 'i';
var.entier = 42;
// On le transforme maintenant en réel 'f', et on lui affecte 42.51
var.typeVariant = 'f';
var.reel = 42.51;
// C'est la même variable en mémoire qui est utilisée, mais son contenu
// n'est plus interprété comme un entier.
// Sans le membre typeVariant, on ne saurait pas quel membre lire.
```

[11] Type-cast : aussi appelé « transtypage ». Désigne la conversion d'une variable vers un type différent, comme un entier vers un réel, ou un nombre vers une chaîne… Le type-cast préserve le sens de la valeur, donc si on a 42 en entier, on obtiendra 42.0 en réel, même si en mémoire ça ne sera plus du tout la même donnée binaire.

Les unions devraient être très rares dans les SDK.

Cependant elles pourraient être pratiques pour traiter la couleur d'un pixel, et donc on risque d'en trouver dans le domaine de la photo et de l'impression. Elles permettent de considérer un pixel de 2 manières : en tant qu'entier 32 bits (couleur complète), ou en tant que tableau de 4 entiers 8 bits (composantes RGBA). Dans WinDev on devrait utiliser RVBRouge(), RVBVert(), RVBBleu().

2.2.7 Callbacks et pointeurs sur fonctions

En C, un pointeur sur fonction permet de stocker l'adresse en mémoire d'une fonction, pour ensuite l'exécuter. L'appel d'une fonction implique les étapes suivantes :

- Ecrire les paramètres au bon endroit dans la mémoire pour que la fonction appelée puisse les trouver. Selon les conventions d'appel, les paramètres ne seront pas dans le même ordre, ou pas au même endroit (registres ou pile).
- Déplacer le pointeur d'exécution pour « sauter » au début de la fonction appelée.
- Quand la fonction appelée remet le pointeur d'exécution dans la fonction appelante, il peut être nécessaire, selon la convention d'appel, de repositionner la pile.

Donc, l'adresse d'une fonction ne suffit pas à l'appeler, il faut savoir :

- Quels paramètres elle attend.
- Quelle est sa convention d'appel, pour lui passer les paramètres correctement et ne pas se prendre un plantage en sortie de fonction à cause d'une pile décalée.

En C, ces informations ne sont que déclaratives, c'est-à-dire qu'on les connaît dans le code source, mais elles sont perdues dans l'exécutable (DLL).

Conséquence : un pointeur sur fonction est typé. Comme on a des pointeurs sur entiers et des pointeurs sur réels, on peut avoir des pointeurs sur « fonction renvoyant un booléen » et des pointeurs sur « fonction prenant un entier en paramètre et sans valeur de retour ».

Exemple :

```
// Déclaration du type func1_t :
// Fonction ne prenant pas de paramètre, et ne retournant rien.
typedef void (*func1_t)(void);

// Déclaration du type func2_t :
// Fonction prenant 2 réels en paramètre, et retournant une chaîne.
// On comprend ici pourquoi le *func est entre parenthèses, sinon ça serait
// une déclaration de fonction retournant un pointeur sur pointeur.
typedef char *(*func2_t)(double, double);

// Voici maintenant la déclaration d'une fonction qui prend en paramètre
// un pointeur sur fonction de type func2_t
void use_callback1(func2_t f);

// Sans utiliser le typedef, on aurait pu écrire directement :
void use_callback2(char *(*f)(double, double));
```

2.3 API Windows

Beaucoup de SDK sont dépendants de l'API Windows, et utilisent des types de données définis dans le header `windows.h` qui donne accès à cette API.

Afin de savoir ce qui se cache derrière ces types `DWORD`, `LPCSTR`, `BOOL`, etc. il faut consulter la page suivante :

http://msdn.microsoft.com/en-us/library/aa383751%28v=vs.85%29.aspx

Ou alors rechercher : « MSDN Windows data types »

3. Equivalences WinDev

Avant de lire cette partie, il faut s'assurer de bien connaître la fonction `API()`.

Il faut notamment comprendre que cette fonction ne connaît pas le prototype de la fonction appelée, elle se fie aux paramètres qu'on lui passe.

Elle retournera toujours un entier système, et c'est à l'appelant d'interpréter correctement cette valeur de retour. En réalité, il existe un cas particulier qui est le retour d'un réel, mais il est très rare et ne peut être géré que dans WD16 ou supérieur.

3.1 Types scalaires

C	WinDev
int, long	Entier
unsigned int, unsigned long	Entier sans signe
short	Entier sur 2
unsigned short	Entier sans signe sur 2
char	Entier sur 1
unsigned char	Entier sans signe sur 1
float	Réel sur 4
double	Réel sur 8
size_t	Entier système
Pointeur (type *)	Entier système
long long	Entier sur 8
unsigned long long	Entier sans signe sur 8

3.2 Pointeurs

Pour récupérer un pointeur dans WinDev, il faut utiliser l'opérateur & :

```
tabTableau est un tableau de 4 entiers = [ 1, 2, 3, 4 ]
sChaîne est une chaîne = "Hello world"
nEntier est un entier = 12
nOctet est un entier sur 1 = 127
nPtr est un entier système

nPtr = &tabTableau  // Pointeur sur le premier élément du tableau
nPtr = &sChaîne     // Pointeur sur le 1er caractère de la chaîne
nPtr = &nEntier
nPtr = &nOctet
```

Attention : un pointeur peut être une donnée volatile.

En interne, une chaîne ou un tableau est un objet composé d'un pointeur et d'un entier indiquant la taille de la zone mémoire allouée à la chaîne ou au tableau.

Quand on agrandit une chaîne ou un tableau, il se peut que la zone mémoire ne soit plus assez grande, et donc WinDev va en allouer une nouvelle. Si l'adresse à laquelle il a pu l'allouer est différente de l'adresse de sa précédente allocation, il va mettre à jour son pointeur, mais tout ça sera invisible pour le développeur WinDev.

Imaginons alors le cas suivant :

```
// Prépare un buffer de 100 caractères, rempli de zéros
sBuffer est une chaîne = Répète(Caract(0), 100)
nPtr est un entier système

// Récupère le pointeur de la chaîne pour le passer au SDK
nPtr = &sBuffer
SI SDKProvideBuffer(nPtr, 100) = ERR_BUFFER_TOO_SMALL ALORS
       // Le SDK trouve que 100 caractères ne suffiront pas,
       // alors on agrandit la chaîne à 1000 caractères
       sBuffer += Répète(Caract(0), 900)
       SDKProvideBuffer(nPtr, 1000)
FIN
```

Ce code risque de planter aléatoirement.

En effet, lors du deuxième appel à la fonction du SDK, on lui passe le pointeur qu'avait la chaîne avant qu'on l'ait agrandie avec une concaténation. Une fois la chaîne agrandie, rien ne garantit que son contenu soit toujours au même endroit en mémoire. Donc le pointeur est invalide et il faut le récupérer à nouveau avec l'opérateur &.

3.3 Types chaîne et tableau fixes

Une chaîne fixe est une chaîne dont la longueur est fixée à la déclaration et ne pourra jamais être modifiée.

Attention : le type « chaîne fixe » en WinDev désigne un type chaîne issu du BASIC où la chaîne est complétée par des espaces. Le vrai type chaîne fixe se déclare sans le mot « fixe ».

Voici des exemples :

```
sChaîne1     est une chaîne sur 10
szChaîne2    est une chaîne ASCIIZ sur 10
tabTableau   est un tableau fixe de 10 entiers
```

sChaîne1 est une chaîne prenant 10 octets en mémoire. Ça veut dire qu'avec le terminateur nul, cette chaîne ne peut contenir qu'un texte de 9 caractères. Si on lui affecte un 10ème caractère, elle deviendra incompatible avec du code C qui s'attend à trouver le terminateur.

szChaîne2 est une chaîne prenant en réalité 11 octets, de manière à contenir le caractère nul. Le problème, c'est que si le SDK a demandé un `char[10]`, ce n'est pas la même chose, notre chaîne est trop longue.

tabTableau est un tableau de 10 entiers, il prend donc 10 * 4 = 40 octets.

3.4 Types chaîne et tableau dynamiques

En WinDev, ces types font un peu de « magie » pour être compatibles avec le C :

Même s'ils ont besoin de plus de choses qu'un simple pointeur, ils sont physiquement stockés en tant que pointeur sur les données. Mais alors comment WinDev trouve-t-il les données supplémentaires dont il a besoin ? Description de la variable, taille du tableau…

L'astuce est que les informations sont positionnées juste avant les données. Et donc le pointeur qui représente le tableau ou la chaîne ne pointe pas réellement sur le début de la zone mémoire appartenant à l'objet, et WinDev n'a qu'à décrémenter ce pointeur pour retrouver la véritable adresse de l'objet.

Un SDK, en revanche, trouvera ce qu'il attend à l'adresse indiquée par le pointeur : les données contenues par la chaîne ou le tableau.

Ce tour de passe-passe permet d'utiliser une chaîne dynamique (taille variable) ou un tableau dynamique dans une structure qui sera ensuite passée à un SDK :

```
ST_Magique est une structure
       sChaîne est une chaîne
       tabTableau est un tableau local d'entiers
FIN
```

Ce code est compatible avec :

```
struct ST_Magique
{
    char *sChaine;
    int *tabTableau;
};
```

Mais attention : le SDK n'a pas accès à toutes les informations que contiennent réellement les tableaux et les chaînes WinDev, et avec la structure décrite plus haut, il ne connaît pas la longueur de chaque. Si on ne lui donne pas l'info, il risque donc de lire ou écrire dans un endroit de la mémoire qui contient tout à fait autre chose, ce qui déclenchera des bugs. Et pour ça, point de magie, il faudra le gérer à la main selon l'interface du SDK.

De plus, si le SDK stocke une copie de la structure pour s'en servir plus tard, il est possible que les pointeurs présents dans `sChaine` ou `tabTableau` ne soient plus valides. On dit que ces pointeurs sont volatiles. Ce cas est expliqué dans une situation plus simple au paragraphe 3.2.

3.5 Types structurés

Une structure WinDev est proche d'une structure C, mais voici les points auxquels il faut faire attention :

```
struct Exemple
{
    char     c;
    char     *chaine1;
    int      *tableau1;
    char     chaine2[10];
    int      tableau2[10];
};
```

- L'alignement du pointeur `chaine1` exige d'allouer un espace entre `c` et `chaine1`.
- `chaine1` et `tableau1` sont des pointeurs, mais `chaine2` et `tableau2` sont des données insérées dans la structure, même si on les manipulera par pointeur.
- De plus, `chaine2` faisant 10 caractères de long, donc 10 octets, il va poser un problème d'alignement pour `tableau2`.

Fort de ces observations, on peut donc écrire le code WinDev suivant :

```
Exemple est une structure
	c			est un entier sur 1
	// Gestion de l'alignement.
	padding1		est une chaîne sur 3
	// Profitons de la magie opérée par WinDev pour le type chaîne.
	chaine1		est une chaîne
	// Pour le tableau, on va se passer de cette magie pour
	// avoir la possibilité de mettre NULL dans certains cas.
	tableau1		est un entier système
	// Voici une chaîne dont les données sont dans la structure
	chaine2		est une chaîne ASCIIZ sur 10
	// Gestion de l'alignement, car le prochain membre est aligné sur 4
	padding2		est une chaîne sur 2
	// Et voici un tableau dont les données sont dans la structure
	tableau2		est un tableau fixe de 10 entiers
FIN
```

3.6 Taille d'une variable en mémoire

Pour connaître la taille réelle d'une variable en mémoire, il faut utiliser différentes fonctions selon le type de données :

```
// Type scalaire, chaîne, structure :
nTaille = Dimension(nEntier)
nTaille = Dimension(sChaîne)
nTaille = Dimension(stStruct)
// Type tableau dynamique :
nTaille = TableauInfo(tabTableau, tiTailleTotale)
```

3.7 Callbacks

Une callback est une fonction destinée à être appelée par la DLL du SDK.

- ➔ J'appelle la fonction `SDKCall()` en lui passant l'adresse de `MaFonction()`
  - ➔ `SDKCall()` appelle `MaFonction()`
    - ➔ `MaFonction()` s'exécute et se termine
  - ← Retour dans `SDKCall()`, qui se termine
- ← Retour dans mon code, qui va pouvoir se terminer à son tour

Pour passer un pointeur sur fonction à un SDK, il faut utiliser l'opérateur & :

```
SDKCall(&MaFonction)
```

Important : il est impératif de respecter le prototype de la fonction définie en C, sans quoi on s'expose à des plantages.

Il existe un semblant de norme indiquant que toutes les callbacks devraient utiliser la même convention d'appel : le stdcall. En conséquence WinDev ne crée que du stdcall. Normalement, aucun SDK ne dérogera à la règle.

3.8 Appels de fonctions

Appeler une fonction de DLL demande un peu de rigueur dans le typage des paramètres.

Pour WD15 et antérieur, il est recommandé d'encapsuler chaque appel de fonction dans une fonction WinDev dont les paramètres sont typés. Afin d'éviter des erreurs W-Langage on déclarera souvent ces paramètres comme étant locaux, ce qui permet des conversions de types implicites :

```
PROCEDURE SDKFunction(LOCAL i est entier, s est chaîne,
                      LOCAL r est réel sur 4, LOCAL st est une ST_Struct)
// i est passée par valeur
// s est passée par adresse
// r est passée par valeur
// st est passée par adresse, mais si le SDK la modifie ça n'aura pas d'effet
// car la fonction WinDev l'a passée par valeur
RENVOYER API("SDK.DLL", "SDKFunction", i, &s, r, &st)
```

La fonction ci-dessus est sécurisée, car elle assure le bon typage des paramètres. Mais pour la structure `st` on a rendu impossible une modification par le SDK, la fonction WinDev ne manipule pas la structure d'origine, mais une copie locale. La chaîne `s`, au contraire, pourra être modifiée et conservera les changements.

Il y a 2 types qui ne peuvent pas être passés par valeur à la fonction API : les chaînes et les tableaux. Donc, même si on n'utilise pas l'opérateur &, WinDev passera en réalité un pointeur sur les données.

Depuis WD16 on peut déclarer une « Description d'API », ce qui est équivalent à l'encapsulation dans une fonction, mais peut-être plus performant et plus sécurisé. Ça permet surtout de gérer plus de cas, préciser la convention d'appel etc.

3.9 Récupération de données

Récupération d'une chaîne ASCIIZ dont on n'a que l'adresse :

```
nPtr est un entier système = SDKGetString()
sChaîne est une chaîne = ChaîneRécupère(nPtr, crAdresseASCIIZ)
```

Récupération d'une structure dont on n'a que l'adresse :

```
nPtr est un entier système = SDKGetStruct()
stStruct est une ST_Struct
```

```
Transfert(&stStruct, nPtr, Dimension(stStruct))
```

Si en plus on est censé modifier des données à l'adresse que le SDK nous a donnée, il faut penser à faire la copie en sens inverse :

```
nPtr est un entier système
stStruct est une ST_Struct
sChaîne est une chaîne

// Remplace le début de la chaîne du SDK par "Toto"
nPtr = SDKGetString()
sChaîne = ChaîneRécupère(nPtr, crAdresseASCIIZ)
sChaîne[[1 A 4]] = "Toto"
Transfert(nPtr, &sChaîne, Dimension(sChaîne))
// Echange x et y dans la structure du SDK
nPtr = SDKGetStruct()
Transfert(&stStruct, nPtr, Dimension(stStruct))
stStruct.x <=> stStruct.y
Transfert(nPtr, &stStruct, Dimension(stStruct))
```

3.10 Charger une DLL

Si on n'appelle pas la fonction ChargeDLL, WinDev chargera et déchargera la DLL à chaque appel à API. Outre la perte de performance, un tel fonctionnement ne permet pas à la DLL de maintenir des données internes, et peut provoquer des bugs. Il faut donc penser à utiliser ChargeDLL.