Institut des Sciences et Techniques de l'Ingénieur d'Angers 2006-2007

Programmes Windows sous Visual C++ avec les MFC

MCours.com

Bertrand Cottenceau MASTER 2 Pro IAIE

1 Introduction

Ce cours aborde la programmation d'applications pour Windows en C++, à l'aide de la bibliothèque **MFC** (***Microsoft Foundation Classes*).** Cette programmation est réalisée dans l'environnement Microsoft Visual C++ qui contribue également à simplifier cette tâche dans la mesure où des assistants (`AppWizard` et `ClassWizard`) génèrent du code. L'essentiel du travail de programmation consiste alors à compléter des classes et des méthodes générées par des assistants.

Les MFC sont une bibliothèque de classes C++ qui encapsulent les différentes fonctions, notamment de gestion des fenêtres ou de messages, propres à Windows. On dit aussi que les classes MFC encapsulent l'API (Application Programing Interface) Windows. Par exemple, la classe **`CWnd`** va encapsuler toutes les fonctions et les structures de données nécessaires à la gestion des fenêtres[1]. Les différentes fenêtres, les menus, les barres d'outils, ou les contrôles d'une application seront donc vus comme des objets instanciés à partir des classes MFC ou de classes dérivées des classes MFC. L'écriture d'un programme Windows va donc consister à créer des classes et à gérer des instances de celles-ci.

En fait, la bibliothèque MFC est une collection hiérarchisée très vaste de différentes classes. En outre, du fait des différentes dérivations, certaines classes ont accès à un nombre très élevé de méthodes (fonctions membres). Il peut donc sembler difficile de développer une application Windows avec les MFC sans avoir une bonne connaissance de cette hiérarchie de classes. C'est pourquoi l'environnement Visual C++ dispose d'un générateur de code (assistant `AppWizard`) fournissant, après quelques étapes de configuration de paramètres (choix application SDI/MDI, gestion ou non d'accès à une bases de données, utilisation ou non de sockets ...), un canevas fonctionnel d'une application contenant par exemple une fenêtre cadre, une fenêtre fille, un menu et une barre d'outils. Ensuite, le programmeur n'a plus qu'à compléter ce canevas pour ajouter des fonctions particulières. D'ailleurs, même l'ajout de fonctionnalités ou de méthodes dans une classe se fait au moyen d'un assistant appelé `ClassWizard`.

Finalement, à l'aide de ces outils, la programmation Windows repose sur la compréhension de quelques principes de base liés à Windows (tels que la notion de programmation événementielle) ainsi que sur la bonne utilisation des différents assistants, ainsi que des différentes fonctionnalités de l'environnement de développement Visual C++ (navigation dans les classes, dans les fichiers, les options de compilation ...). Pour s'en convaincre, on peut d'ailleurs faire rapidement un premier essai.

Exercice pratique : sélectionner `File|New|onglet Projects| MFC AppWizard(.exe)`
Choisir un nom pour le projet : « **PremierExple** » **,** puis cliquer sur **OK.** Dans l'étape 1 de AppWizard, cliquer sur **Finish.** On obtient ainsi un canevas d'application que l'on peut **compiler et exécuter.** Il s'agit d'une application MDI (Multi Document Interface) contenant un menu et une barre d'outils.

2 Rappels de C++

La bibliothèque MFC est une bibliothèque de classes C++. Il convient donc de maîtriser un certain nombre de notions de base du C++ pour pouvoir comprendre et exploiter le code généré par les assistants de l'IDE Visual C++. On va donc faire quelques rappels sur le langage C++ et apporter des précisions sur certaines techniques utilisées par le framework MFC, notamment en ce qui concerne l'exploitation de macros développées par le préprocesseur.

Les pages suivantes contiennent des fiches synthétiques concernant l'écriture de classes.

[1] Nous verrons que sous Windows la notion de fenêtre est assez large dans la mesure où les contrôles (bouton, zones d'éditions de texte, images ...) sont également des fenêtres.

2.1 Fiche 1 : Les classes en C++

Déclaration d'une classe (dans un fichier header d'extension .h)

```
class MaClasse
{
	public:
		// méthodes à accès public
		MaClasse();			// constructeur sans argument
		MaClasse(int var);		// constructeur à un argument de type int
		MaClasse(const MaClasse & obj);	// constructeur de copie
		~MaClasse();				// destructeur

		MaClasse & operator=(const MaClasse & obj); // opérateur d'affectation

		void SetVal(int val);  // Méthode modificateur d'état, donc non constante

		int GetVal() const;   // Méthode pour obtenir tout ou partie de l'état
				      // donc méthode constante
private:
		// données / méthodes à accès privé
		int _val;		// données membres encapsulées (à accès privé)
		double _autre;
};
```

Définition des méthodes dans un fichier séparé (fichier d'extension .cpp) [source partielle]

```
MaClasse::MaClasse()
{
	_val=0;		// la donnée membre _val est initialisée à 0;
	_autre = 2.1;
}

MaClasse::MaClasse(const MaClasse & obj)
{
	// implementation non fournie
	...
}

int MaClasse::GetVal() const
{
	return _val;   // retourne la valeur de la donnée membre _val(l'état de l'objet)
}

void MaClasse::SetVal(int val)
{
	_val=val;	// affecte une nouvelle valeur à l'état
}
```

A noter

- un constructeur/un destructeur ne retourne rien (ne pas mettre void non plus)
- le destructeur n'a pas d'argument
- un constructeur sert à donner un état cohérent à l'objet (penser à initialiser toutes les données membres)
- un accesseur est une méthode qui retourne ou donne accès à l'état de l'objet. C'est une méthode constante.
- inversement, une méthode qui permet la modification de l'état d'un objet ne doit pas être constante.

Utilisation de la classe MaClasse

```
void main()
{
	MaClasse objet1;	// construction d'un objet à l'aide du
	// constructeur sans argument
	// etat = (objet1._val==0, objet1._autre==2.1)

	MaClasse objet2(15);  // utilise le constructeur avec 1 argument int pour objet2

	MaClasse objet3(objet1); // utilise le constructeur par copie pour objet3

	cout << objet1.GetVal() << endl;

	objet1.SetVal(12);	// nouvel état : (objet1._val==12,objet1._autre==2.1)
}
```

2.2 Fiche 2 : Gestion des projets avec des classes en C++

Pour chaque classe, on place dans un fichier header (extension .h) la définition de la classe, c'est-à-dire son nom, les noms et types des arguments et les prototypes des méthodes.

On place dans un fichier source (extension cpp) la définition des méthodes.

Enfin, pour utiliser la classe, il faut créer un projet incluant le fichier source d'utilisation de la classe (fichier cpp) et le (ou les) fichier(s) de définition des méthodes. Le fichier header doit être visible par tous les fichiers qui l'utilisent

Fichier header (fichier d'entête)

```
---------------------------------------------------------------------------
// fichier d'entête maclasse.h
#ifndef __MA_CLASSE__
#define __MA_CLASSE__        // directives pour compilation conditionnelle

#include <iostream>  // inclusion des fichiers nécessaires ici

class MaClasse
{
      public:
              MaClasse();

              void Affiche(std::ostream & sortie) const;
};
#endif
---------------------------------------------------------------------------
```

Fichier de définition des méthodes (implémentation de la classe)

```
---------------------------------------------------------------------------------------------
// fichier maclasse.cpp
#include "maclasse.h"  // le fichier header doit être dans le même répertoire que ce source

MaClasse::MaClasse()
{
      //...
}

void MaClasse::Affiche(std::ostream & sortie) const
{
      sortie << "Affichage sur sortie standard";
}
---------------------------------------------------------------------------------------------
```

Fichier d'utilisation de la classe

```
---------------------------------------------------------------------------------------------
// utilisation.cpp
#include "maclasse.h"        //  le  fichier  header  doit  être  dans  le  même  répertoire  que  ce
source

void main()
{
      MaClasse m;
      m.Affiche(std::cout);         //affiche sur flot cout
}
---------------------------------------------------------------------------------------------
```

Projet C++ : pour pouvoir compiler et éditer les liens, les deux fichiers `maclasse.cpp` et `utilisation.cpp` doivent être dans un même projet.

Note : on peut également faire un fichier librairie à partir d'un ou de plusieurs fichiers de sources de définition des méthodes d'une classe. Pour utiliser une classe, l'utilisateur n'a plus ensuite qu'à créer un projet incluant la librairie (à la place des fichiers sources de définition des méthodes).

2.3 Fiche 3 : Introduction aux classes standard (STL) et aux classes paramétrées en type

Les classes standards `istream`, `ostream`, `string`, `vector<T>`, `list<T>` utilisent l'espace de nom **`std`** :

```
#include <iostream>          // déclaration des classes de flots
#include <string>            // déclaration de la classe de chaînes de caractères

using namespace std;         // utiliser l'espace de nom std

void main()
{
       cout << "Hello world !" << endl;

       string str1("abc");   // construction d'un objet string
       string str2("def");
       string str3(str1);    // str3 est construit par copie de str1

       str1=str1+str2;       // l'opérateur + réalise une concaténation

       cout << str1 << endl; // un objet string s'affiche sur le flot cout

       cout << "3ième caractère de str1 :"
cout << str1[2] << endl; // un objet string s'utilise comme un tableau
}
```

Utilisation d'une classe paramétrée en type.

La bibliothèque STL fournit des classes paramétrées en type pour stocker des données

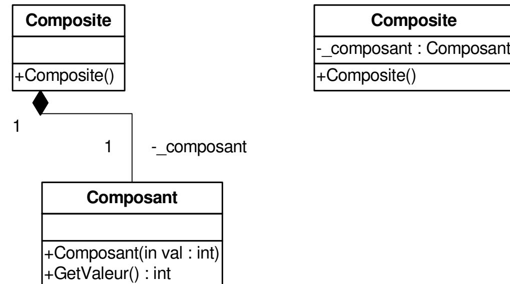
```
       vector<Ty>     = classe de tableaux stockant des données de type Ty
       stack<Ty>      = classe de piles de données de type Ty
```

```
---------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>            // déclaration de la classe paramétrée vector<Ty>

using namespace std;         // utiliser l'espace de nom std aussi pour ces classes

void main()
{
       vector<int> v1(5,2);  // v1 est un vecteur de 5 int initialisés à 2.
       vector<int> v2(8);    // v2 est de taille 8, à valeurs initiales nulles

       v1[2]=3;       // v1 s'utilise comme un tableau
       v1[4]=2;

       // ici, éviter v1[5]=7 car les indices valides vont de 0 à 4 (car la taille est 5)

       v1.resize(12);        // méthode qui modifie la taille du vecteur

       cout << v1.size() << endl;    // vector<Ty>::size() est l'accesseur de taille

       v1[11]=4;      // ici c'est bon, car le vecteur vient d'être redimensionné

       v2=v1;         // l'affectation entre deux objets vector<int> est possible

       vector<char> v3(7);   // v2 est un vecteur de caractères de taille 7

       v2[0]='a';     // v3 stocke donc des caractères

       stack<int> pile;      // objet "pile" pour gérer des int

       pile.push(12);        // place la valeur 12 sur la pile

       stack<float> pileF;   // objet "pile" pour gérer des float

       pileF.push(3.25);     // place la valeur float 3.25 sur la pile pileF

       /* ce qui ne fonctionne pas (voir chapitre dédié aux classes paramétrées)

       v3=v1;         // les classes vector<int> et vector<char> sont différentes

       */
}
---------------------------------------------------------------------------------------------
```

2.4 Fiche 4 : Réalisation de la relation de composition en C++

Certaines classes ont comme attributs des objets d'une autre classe. On représente cela en UML soit par la relation de composition ou bien comme le fait que certains attributs sont d'un type classe donné.

Ci- contre, deux représentations de la même classe `Composite` sont données. La représentation de gauche indique que la classe `Composite` est composée d'un objet `_composant`. La représentation de droite indique qu'un attribut (`_composant`) de la classe `Composite` est de type `Composant`.



Dans les deux cas, une réalisation possible en C++ de la classe `Composite` est la suivante

```
------------------------------------------------------------------------
//fichier composite.h
#ifndef __CLS_COMPOSITE__
#define __CLS_COMPOSITE__

#include "composant.h"

class Composite
{
public:
      Composite();
      Composant GetComposant() const;  // retourne une copie du composant
private:
      Composant _composant;         // objet membre
};
#endif
------------------------------------------------------------------------
```

Définition des méthodes dans un fichier séparé (fichier d'extension .cpp)

```
------------------------------------------------------------------------
#include "composite.h"
Composite::Composite():_composant(12)
{
}

Composant Composite::GetComposant() const
{
      return _composant;
}
------------------------------------------------------------------------
```

Il faut souligner le rôle de la *liste d'initialisation* pour les classes composites. Ci-dessus, l'objet composant est initialisé par le constructeur à 1 argument de type int (seul constructeur disponible pour cette classe).

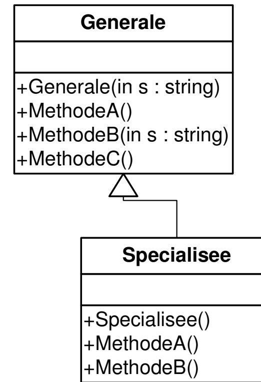
```
------------------------------------------------------------------------
#include "Composite.h"

void main()
{
      Composite c1;

      Composant c2(c1.GetComposant());

}
------------------------------------------------------------------------
```

Remarque : pour l'exemple ci-dessus, la classe Composant doit être dotée d'un constructeur de copie valide : d'une part pour que l'accesseur Composite::GetComposant() retourne une copie du composant et que la construction de c2 soit possible dans le programme d'utilisation.

2.5 Fiche 5 : Réalisation de la relation de spécialisation en C++

Une classe décrit ce que les objets d'un ensemble ont en commun en termes de méthodes et d'attributs. D'un point de vue ensembliste, une sous-classe caractérise donc un sous-ensemble d'objets. Soit cette sous-classe a des attributs supplémentaires permettant d'apporter une représentation plus précise des objets, soit elle se distingue par le fait que les attributs ne peuvent prendre que certaines valeurs. La relation sous-classe/super-classe ou relation de spécialisation/ généralisation se représente en UML par une flèche. La flèche part de la sous-classe (ou classe spécialisée) et désigne la super-classe (la classe la plus générale).

```
---------------------------------------------
// fichier header de la classe Specialisee
#ifndef __CLS_SPECIALISEE__
#define __CLS_SPECIALISEE__

#include "generale.h"

class Specialisee : public Generale
{
public:
	Specialisee();
	void MethodeA();
	void MethodeB();
};

#endif
---------------------------------------------
```



Définition des méthodes dans un fichier séparé (fichier d'extension .cpp)

```
-----------------------------------------------------------
// définition des méthodes de la classe Specialisee
#include "specialisee.h"
Specialisee::Specialisee():Generale(const std::string & s)
{
}

void Specialisee::MethodeA()
{
	// masque la méthode de la super-classe
	// ...
}
-----------------------------------------------------------
```

Là encore, il faut souligner le rôle de la liste d'initialisation. Ci-dessus, l'objet spécialisé initialise la partie héritée par le constructeur à 1 argument de type string. On peut supposer qu'ici la classe générale contient un objet membre de type string, mais ça importe peu. Ce qui importe c'est de regarder l'interface de la super-classe pour savoir quels sont les constructeurs disponibles.

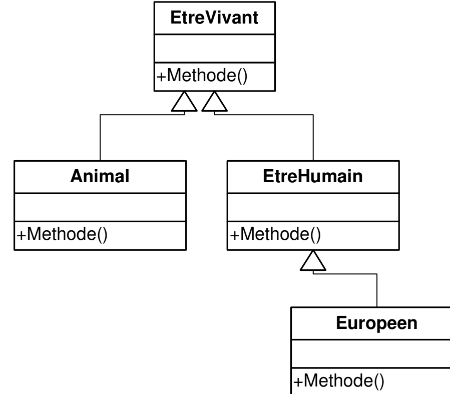
Toutes les méthode publiques de la super-classe sont utilisables sur la sous-classe. Eventuellement, certains phénomènes de masquage nécessitent l'utilisation du nom de la classe et de l'opérateur de portée ::

```
-----------------------------------------------------------
#include "specialisee.h"

void main()
{
	Specialisee s1;

	s1.MethodeC();  // la méthode C est accessible pour la sous-classe

	s1.MethodeB();        // méthode B de la super-classe
	s1.MethodeB(std::string("toto"));    // méthode B de la sous-classe

	s1.MethodeA();        // la méthode de la classe spécialisée
	                      // masque celle de la super-classe

	s1.Generale::MethodeA();     // en raison du masquage, il faut préciser
	                             // si l'on souhaite
	                             // utiliser la méthode de la super-classe.
}
-----------------------------------------------------------
```

2.6 Fiche 6 : Le polymorphisme en C++

Lorsqu'une hiérarchie de classes existe, le C++ autorise certaines conversions de type. D'un point de vue ensembliste, il est normal que les objets d'une sous-classe puissent être aussi considérés comme des objets de la super-classe. Par exemple, la classe des êtres humains étant un sous-ensemble de celle des êtres vivants, un être humain **est un** être vivant.

```
class EtreVivant
{
public:
      void Methode();
};

class Animal: public EtreVivant
{
public:
      void Methode();
};

class Europeen: public EtreHumain
{
public:
      void Methode();
};
```



Pour la hiérarchie décrite ci-dessus, certaines conversions de types sont possibles. Par ailleurs, faute de précision, la ligature des méthodes est *statique*. Cela signifie que via un pointeur de type `EtreHumain *`, on n'accède qu'aux méthodes de la classe `EtreHumain` ou des super-classess (par exemple `EtreVivant`).

```
void main()// conversion sous-classe * -> classe *
{
Europeen E;
Animal A;
EtreHumain * ptrEH=&E;       // Europeen * -> EtreHumain *

ptrEH->Methode();     // appel de la méthode de la classe EtreHumain même si l'objet
                      // pointé est de type Europeen

ptrEH->EtreVivant::Methode();        //méthode également accessible

EtreVivant * ptrEV = ptrEH;  // conversion EtreHumain * -> EtreVivant *

ptrEV->Methode();     // appel de la méthode de la classe EtreVivant

EtrVivant * ptrG2 = &A;      // Animal * -> EtreVivant *
}
```

Ligature dynamique : on peut préciser dans la classe de base si une recherche dynamique de méthode doit être effectuée (mot clé virtual).

```
class EtreVivant
{
public:
virtual void Methode();      // demande au compilateur de mettre en place un lien dynamique
virtual ~EtreVivant();       // le destructeur doit être virtuel
};

void main()
{
      EtreVivant * tableau[4];
      tableau[0]=new EtreVivant;
      tableau[1]=new Animal;
      tableau[2]=new Europeen;
      tableau[3]=new EtreHumain;

tableau[0]->Methode();       // méthode de la classe EtreVivant
tableau[1]->Methode();       // méthode de la classe Animal
tableau[2]->Methode();       // méthode de la classe Europeen
tableau[3]->Methode();       // méthode de la classe EtreHumain

for(unsigned i=0;i<4;i++) delete tableau[i];
}
```

2.7 Fiche 7 : Les membres statiques

Une classe peut contenir des données membres statiques et des méthodes statiques. Une donnée membre statique n'existe qu'en un seul exemplaire pour tous les objets (à la différence des attributs). C'est une donnée dépendant de l'espace de nom de la classe et partagée par les objets de la classe. Une fonction statique est une fonction membre qui ne peut accéder qu'aux membres statiques. Une telle méthode n'a pas de membre this et peut être appelée indépendamment de tout objet.

On peut ainsi compter le nombre d'objets instanciés dans une classe (compteur d'instances).

```
----------------------------------------------------------------------------
// classeA.h
#ifndef __CLASSEA__
#define __CLASSEA__

class ClasseA
{
private:
	static unsigned _nbObjets;    // compteur d'instances de la classe
public:
	ClasseA();
	~ClasseA();

	static unsigned GetNombreInstances() const;
};

#endif
----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
// classeA.cpp
#include "classeA.h"

unsigned ClasseA::_nbObjets=0;      // definition et initialisation du membre statique

ClasseA::ClasseA()
{
	_nbObjets++;
}

ClasseA::~ClasseA()
{
	_nbObjets--;
}

unsigned ClasseA ::GetNombreInstances() const
{
	return _nbObjets ;
}
----------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------
#include "classeA.h"
#include<iostream>

void main()
{
	ClasseA A1,A2;
	std::cout << ClasseA::GetNombreDInstances();  // affiche 2
	Classe A3 ;
	std::cout << ClasseA::GetNombreDInstances();  // affiche 3
}
----------------------------------------------------------------------------
```

2.8 Fiche 8 : Les patrons de classes en C++

En C++, on peut introduire des paramètres de type. Pour les classes paramétrées, il est préférable de placer la définition des méthodes (elles-mêmes paramétrées) dans la définition de la classe. Le fichier header (.h) contient alors toute l'implémentation. Il n'y a pas de fichier de définition cpp pour les patrons de classe. Exemple d'un patron de conteneurs de type tableau (accès direct aux éléments).

```
-------------------------------------------------------------------------------
#ifndef __CLSPARAM__
#define __CLSPARAM__

template<class T>
class Tableau
{
public:
       Tableau<T>(unsigned size=1)
       {
              _tab=new T[size];
              _size=size;
       }

       Tableau<T>(const Tableau<T> & t)
       {
              _tab=new T[t._size];
              _size=t._size;
              for(unsigned i=0;i<_size;i++) _tab[i]=t._tab[i];
}

Tableau<T>& operator=(const Tableau<T> & t)
       {
              T* ptr=_tab;
              _tab=new T[t._size];
              _size=t._size;
              for(unsigned i=0;i<_size;i++) _tab[i]=t._tab[i];
              delete [] ptr;
              return *this;
}

~Tableau<T>(){ delete [] _tab;}

void GetElement(unsigned i) const { return _tab[i];}
void SetElement(unsigned i, const T & valeur) { _tab[i]=valeur; }

unsigned GetSize() const { return _size;}

T& operator[](unsigned i) {  return _tab[i]; }
const T &  operator[](unsigned i) const { return _tab[i]; }

private:
       T * _tab;             // T est le paramètre de type
       unsigned _size;
};
#endif
-------------------------------------------------------------------------------
```

Utilisation du patron :

```
-------------------------------------------------------------------------------
#include "Tableau.h"
#include "Rationnel.h"
#include "Point.h"


void main()
{
       Tableau<int> tab1(3);
       Tableau<Rationnel> tab2(4);
       Tableau<Point> tab3(10);

       tab1.SetElement(2,3);
       tab2.SetElement(0,Rationnel(1,2));
       tab1[0]=4;
}
-------------------------------------------------------------------------------
```

2.9 Le RTTI (Run Time Type Information) et les nouvelles syntaxes de cast

En C, le transtypage (cast) utilise la syntaxe suivante :

```
float f=2.5 ;
unsigned short * pS=(unsigned short *) &f;    // float * -> unsigned short *

// si la classe Derivee dérive de Base
Base * pB=new Derivee;                        // Derivee * -> Base *
((Derivee *)pB)->MethodeDeDerivee();          // Base * -> Derivee *
delete pB;
```

En C++, on peut utiliser la même notation ou utiliser de nouvelles syntaxes.

```
float f=2.5 ;

// quand il n'y a pas de lien entre les types
unsigned short * pS=reinterpret_cast<unsigned short *>(&f);

// pour naviguer dans une hiérarchie de classes
Base * pB=new Derivee;

// ici, on sait que pB désigne en fait un objet de la classe dérivée
static_cast<Derivee *>(pB)->MethodeDeDerivee();
```

Une troisième forme de cast (**`dynamic_cast<>()`**) repose sur la reconnaissance dynamique de type (RTTI).

Les méthodes polymorphes (déclarées virtuelles dans une des super-classes) peuvent être retrouvées dynamiquement à l'exécution. La reconnaissance dynamique de type à l'exécution (RTTI) repose sur le polymorphisme. D'ailleurs, le RTTI ne fonctionne que sur les classes polymorphes, c'est-à-dire possédant au moins une méthode virtuelle. Les classes polymorphes peuvent utiliser l'opérateur `typeid()` qui retourne un objet de la classe `type_info`, ce dernier permet d'identifier la classe, notamment son nom.

Considérons une classe `Base` (ayant une méthode virtuelle) et deux classes dérivées `DeriveeA` et `DeriveeB`. Le programme suivant peut alors être exécuté (RTTI doit être activé au niveau du compilateur) :

```
void main(){
DeriveeA DA ;
DeriveeB DB ;
Base B;
      cout << typeid(DA).name() << endl;      // affiche DeriveeA
      cout << typeid(DB).name() << endl;      // affiche DeriveeB
      cout << typeid(B).name() << endl;       // affiche Base

      Base * pB=new DeriveeA;
      cout << typeid(*pB).name();      // affiche DeriveeA
      if(typeid(*pB)==typeid(DA)) cout << "DA et *pB sont de la même classe" ;
      delete pB ;

      vector<Base*> v ;
      v.push_back(new Base) ;
      v.push_back(new DeriveeB);
      v.push_back(new DeriveeA);
      DeriveeA * pA;

      for(unsigned i=0;i<v.size();i++){
            pA=dynamic_cast<DeriveeA *>(v[i]);
            //pA==NULL si (*pA) n'est pas de type DeriveeA
            if(pA!=NULL) pA->MethodeDeDeriveeA();
            delete v[i];
      }
}
```

2.10 Le préprocesseur

La compilation d'un fichier source s'effectue en plusieurs phases. La première est celle du préprocesseur. Le préprocesseur travaille à partir des directives #____. Il réalise des modifications lexicales telles que l'inclusion textuelle (#include) ou la définition de macros (texte remplacé par un autre).

Inclusion textuelle : la directive **`#include`** permet l'inclusion de fichiers.

```
#include<afx.h>     // recherché dans le répertoire Include du compilateur
#include "ClassA.h" // recherché en premier dans le repertoire courant
```

Compilation conditionnelle : permet d'éviter la définition multiple d'une classe due à des inclusions imbriquées

```
//ClassA.h
#ifndef __MACLASSE__        // si la macro n'est pas définie
#define __MACLASSE__        // définir une macro vide

       class ClassA
       {
              …      // définition de la classe
       };
#endif
```

Les macros : le préprocesseur remplace les occurrences de la macro par un autre texte

```
#define f(a) fonction(a)         // macro paramétrée

       void fonction(unsigned);

void main()
{
       unsigned var=1;
       f(var);             // après développement -> fonction(var);
}
```

Opérateur # : transforme un paramètre de macro en chaîne

```
#define PRINT(a) cout << #a << "=" << (a) << endl

void main()
{
       unsigned var=1;
       std::string str("abc");
       PRINT(var);         // -> cout << "var" << "=" << (var) ;
       PRINT(str);         // -> cout << "str" << "=" << (str) ;
}
```

Compléter une classe via des macros : les MFC proposent des macros pour ajouter des méthodes et des membres à des classes. L'exemple suivant propose deux macros permettant à une classe de retourner son nom sous forme d'un objet string (sans utiliser le RTTI). Le nom de la classe est le paramètre de la macro.

Opérateur ## : concatène   **Opérateur ** : permet de continuer une macro sur la ligne suivante

```
// macros.h
#define DECLARE_NAME(cls) \
       private: \
       static std::string cls##Name;\
       public: \
       virtual std::string GetClassName() const;

#define IMPLEMENT_NAME(cls) \
std::string cls::cls##Name= #cls; \
std::string cls::GetClassName() const \
{ return cls##Name;}
```

Utilisation des macros `DECLARE_NAME(CLS)` et `IMPLEMENT_NAME(CLS)`

```
// classeA.h
#include "macros.h"

class ClasseA
{
   DECLARE_NAME(ClasseA)
public:
   ClasseA();
};
```

```
//classeA.cpp
   IMPLEMENT_NAME(ClasseA)

ClasseA::ClasseA()
{
}
```

Après développement des macros on obtient :

```
// classeA.h
#include "macros.h"

class ClasseA
{
private:
      static std::string ClasseAName;
public:
      virtual std::string GetClassName() const;

public:
      ClasseA();
};
```

```
//classeA.cpp

std::string ClasseA::ClasseAName= "ClasseA";

std::string ClasseA::GetClassName() const
{
       return ClasseAName;
}

ClasseA::ClasseA()
{
}
```

Le développement des macros enrichit la classe d'un membre statique (le nom de la classe) et d'une fonction virtuelle permettant d'obtenir ce nom.

3 Environnement de développement Visual C++

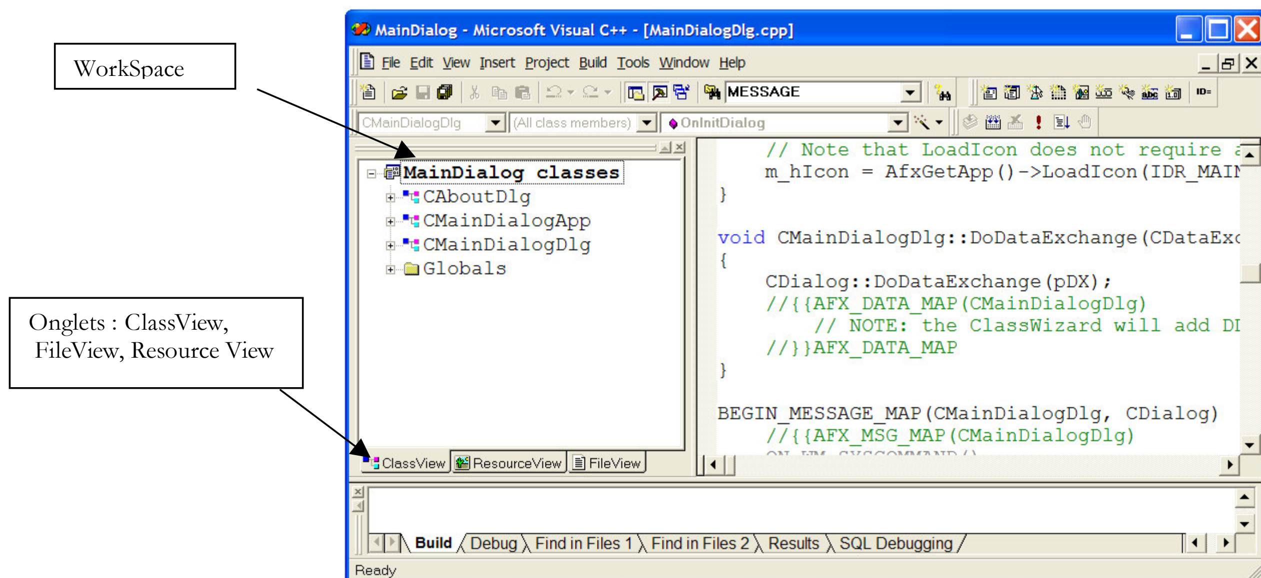
Du fait du grand nombre de classes (et de fichiers) que peut contenir un projet utilisant les MFC, il est nécessaire de bien utiliser certaines fonctionnalités de l'environnement de développement Visual C++. En particulier, la fenêtre Workspace (View/Workspace), normalement visible dès la création d'un nouveau projet, permet la navigation dans les classes/méthodes et dans les fichiers d'un projet. Dans le Workspace, l'onglet **`File View`** fournit une arborescence des différents fichiers du projet, classés par catégorie : fichiers source (.cpp), les fichiers d'entête (.h), les fichiers de ressources (.rc) et les dépendances externes qui notifient les inclusions de fichiers d'entête dans les fichiers sources.

L'onglet **`Class View`** décrit les différentes classes et leurs membres (données/fonctions). On y voit les méthodes et les données membres des classes du projet. Pour qu'une classe soit visible dans cet onglet, le fichier d'entête de la classe doit faire partie des fichiers `headers files` dans l'onglet `File View`. Cet arbre permet facilement la navigation dans les classes du projet. En double-cliquant sur le nom d'une méthode, l'éditeur de texte ouvre et pointe directement sur la définition de la méthode sélectionnée (c'est normalement un fichier source .cpp). En

revanche, en double-cliquant sur le nom d'une classe, on accède directement à la définition de la classe (normalement faite dans un un fichier d'entête .h). Par ailleurs, par un click-droit sur le nom d'une classe, on peut également lancer des utilitaires pour ajouter des données membres ou des méthodes. Lors d'ajout d'une méthode, l'utilitaire ajoute l'entête de la méthode dans la définition de la classe et le corps de la méthode dans le fichier source .cpp correspondant.

Enfin, l'onglet **`Resource View`** donne accès aux différentes ressources d'une application Windows. Ces ressources décrivent l'apparence graphique des fenêtres (tailles, positions des contrôles ..), boîtes de dialogues, menus etc.

Par défaut, tous les fichiers d'un projet sont dans un même répertoire portant le nom du projet comme nom. Dans ce répertoire, un sous-répertoire `res` contient les ressources du projet et un sous-répertoire contient le code objet des fichiers sources compilés ainsi que le fichier exécutable en cas de succès à la compilation et à l'édition des liens.



4 Notion de programmation événementielle sous Windows - API Windows

La programmation en exploitant les MFC utilise l'approche objet en C++. A titre d'exemple, si l'on regarde le projet réalisé en introduction (exercice pratique), une classe `CAboutDlg`, dérivée de `CDialog`, est associée à la boîte de dialogue "A propos de ...". Dès lors, une boîte de dialogue apparaît dès que l'on instancie un objet de la classe `CDialog` puis que l'on invoque la méthode `CAboutDlg::DoModal()`. C'est exactement ce que fait la méthode `CPremierExpleApp::OnAppAbout()` invoquée lors d'un appui sur le bouton ? dans la barre d'outils.

Mais l'approche objet n'est pas indispensable pour réaliser un programme pour Windows. On peut également écrire un programme Windows en langage C (sans les MFC). Une telle programmation repose alors sur l'utilisation de **l'API Windows** (Application Programing Interface) en langage C. L'API est simplement l'ensemble de fonctions du système d'exploitation Windows : gestion des fenêtres (création, restauration…), des messages, des threads, des fichiers, des interfaces graphiques...

• Programmation avec l'API Windows.

Nous allons analyser un court exemple de programme Windows en langage C. Cet exemple va mettre en évidence la notion de programmation événementielle.

// exemple de programme utilisant l'API Windows en C

```
#include<windows.h>
#include<stdio.h>

LONG FAR PASCAL WndProc(HWND,UINT,UINT,LONG);

int PASCAL WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{//début WinMain
        static char szAppName[]="mon programme";
        HWND hWnd;
        MSG msg;
        WNDCLASS wndclass;

        if(!hPrevInstance)
        {
                //style classe de fenêtre
                wndclass.style=CS_HREDRAW|CS_VREDRAW;
                wndclass.lpfnWndProc=WndProc;
                wndclass.cbClsExtra=0;
                wndclass.cbWndExtra=0;
                wndclass.hInstance=hInstance;
                wndclass.hIcon=LoadIcon(NULL,IDI_APPLICATION);
                wndclass.hCursor=LoadCursor(NULL,IDC_ARROW);
                wndclass.hbrBackground=(HBRUSH) GetStockObject(WHITE_BRUSH);
                wndclass.lpszMenuName=NULL;
                wndclass.lpszClassName=szAppName;

                // enregistrement de la classe de fenêtre
                RegisterClass(&wndclass);
        }

        hWnd=CreateWindow(szAppName,
                "Utilisation de l'API Windows",
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,
                0,
                CW_USEDEFAULT,
                0,
                NULL,
                NULL,
                hInstance,
                NULL);

        ShowWindow(hWnd,nCmdShow);
        UpdateWindow(hWnd);

        while(GetMessage(&msg,NULL,0,0))   // Boucle des messages
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }

        return(msg.wParam);
}// fin WinMain

//Fonction de fenêtre
LONG FAR PASCAL WndProc(HWND hWnd,UINT Message,UINT wParam,LONG lParam)
{//début WndProc
        HDC hDC;
        PAINTSTRUCT ps;
        RECT taille_fenetre;
                char chaine[30];
                char affiche[40]="(";

        switch(Message)
        {
        case WM_PAINT:
                hDC = BeginPaint(hWnd,&ps);
                GetClientRect(hWnd,&taille_fenetre);

                itoa(taille_fenetre.bottom,chaine,10);
                strcat(affiche,chaine);
                strcat(affiche,",");
                itoa(taille_fenetre.right,chaine,10);
                strcat(affiche,chaine);
                strcat(affiche,")");

                DrawText(hDC,affiche,-1,&taille_fenetre,
                        DT_SINGLELINE | DT_CENTER  |DT_VCENTER);
                EndPaint(hWnd,&ps);
                break;
```

```
        case WM_DESTROY:
                PostQuitMessage(0);
                break;

        case WM_RBUTTONDOWN:
                ShowWindow(hWnd,SW_SHOWMAXIMIZED );
                break;
        default:
                return DefWindowProc(hWnd,Message,wParam,lParam);
        }
        return NULL;
}// fin WndProc
```

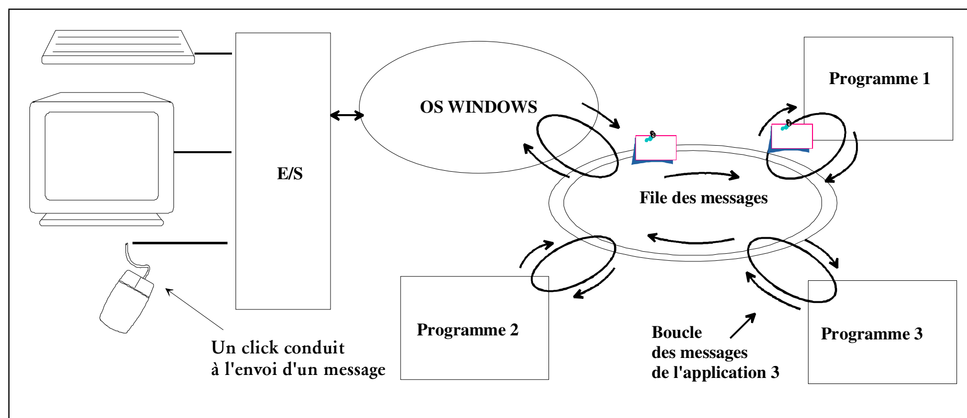
Fonction `WinMain()` : cette fonction est le point d'entrée de l'application Windows, de même que la fonction `main()` est le point d'entrée d'un programme DOS (ou bien en mode console sous Windows).

Le programme principal crée une fenêtre puis lance la *pompe à messages*. Cette « pompe » prélève des messages dans la file de messages. C'est la fonction WndProc qui est utilisée pour réaliser un traitement selon le type de message prélevé. Sur cet exemple, la fonction WndProc ne traite que quelques messages spécifiques, notamment le click droit sur la souris.

Que représente la pompe à messages ?

```
        while(GetMessage(&msg,NULL,0,0))    // pompe à messages
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
```

C'est le principe même du fonctionnement de Windows. Tous les événements liés aux périphériques (clicks souris, événements clavier,..) ou liés à la gestion des fenêtres (une fenêtre vient d'être déplacée, une fenêtre est réduite ...) sont convertis par Windows en **messages**. Ces messages sont diffusés à travers une **file de messages.** La pompe à messages de chaque application **prélève** les messages dans cette file de messages.



Chaque message contient un identifiant de message (une valeur numérique représentée par une constante), et éventuellement des données complémentaires. Par exemple, le click-gauche souris déclenche l'envoi d'un message `WM_LBUTTONDOWN`, dans ce message figurent les coordonnées du click et l'état des boutons de la souris et des touches shift et ctrl du clavier.

Après prélèvement d'un message, l'application peut choisir (dans la fonction `WndProc`) de connecter un traitement à ce message au moyen d'un switch :

```
        switch(Message)
        {
        case WM_PAINT: //gestionnaire sur le message WM_PAINT "repeindre la fenêtre"

        case WM_DESTROY: // message reçu quand la fenêtre est fermée

        case WM_RBUTTONDOWN: //gestionnaire du click droit

        }
```

Le traitement associé à un message est appelé **gestionnaire de message** ou **gestionnaire d'événement**, puisque les événements sur les périphériques sont finalement convertis aussi en messages. On donne ci-dessous quelques messages couramment exploités dans une application :

Messages souris : `WM_LBUTTONDOWN (bouton gauche enfoncé), WM_LBUTTONUP (bouton gauche relaché), WM_LBUTTONDBLCLK (double click gauche), WM_RBUTTONDOWN, WM_RBUTTONUP, WM_RBUTTONDBLCLK, WM_MOUSEMOVE (déplacement de souris)`

Messages clavier : `WM_KEYDOWN (touche non système pressée), WM_KEYUP (touche non système relachée), WM_SYSKEYDOWN (touche système pressée), WM_SYSKEYUP (touche système relachée)`

En résumé, les programmes Windows réagissent à certains événements auxquels ils associent une réponse (un gestionnaire de message). Cette programmation est par conséquent dite « **Evénementielle** ». On peut d'ailleurs observer ces événements au moyen du logiciel **Spy++** fourni avec Visual C++ (section Tools).

Exercice pratique : lancer l'application Spy++ du groupe des programmes "Outils Microsoft Visual Studio". Il s'agit d'un programme capable d'espionner le trafic de messages dans Windows, en faisant éventuellement des restrictions sur les fenêtres analysées et les types de messages espionnés.

5 Introduction aux Microsoft Foundation Classes (MFC) (sans AppWizard ni ClassWizard)

Dans cette partie, nous analysons rapidement quelques exemples d'applications créées à l'aide des MFC sans l'aide des assistants de Visual C++.

5.1 Premier exemple utilisant les MFC

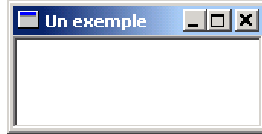
Dans l'exemple ci-dessous, l'application est créée en utilisant les classes `CWinApp` et `CFrameWnd` des MFC.

```
#include <afxwin.h>

class CMaFenetre:public CFrameWnd
{
public:
        CMaFenetre()
        {
                Create(0,"Un exemple");
        }
};

class CMonApp:public CWinApp
{

public:
        virtual BOOL InitInstance()
        {
                m_pMainWnd=new CMaFenetre;
                m_pMainWnd->ShowWindow(m_nCmdShow);
                return TRUE;
        }
};

CMonApp ObjetApplication;
```

Pour pouvoir exécuter ce programme Windows utilisant les MFC (sans l'assistant AppWizard) il faut suivre les étapes suivantes.

- créer un nouveau projet (File|New|onglet Projects|**Win32Application**)
- saisir le code précédent dans un nouveau fichier source (.cpp)
- modifier Project|Settings|General : **Use MFC in a shared DLL**
- compiler, éditer les liens et exécuter.

L’exécution de ce programme conduit à une application avec la fenêtre cadre suivante :



Que remarque-t-on sur cet exemple ?

La classe **`CMonApp`** est dérivée de la classe **`CWinApp`**. Il s’agit de la classe de l’application. Un objet de la classe **`CMonApp`** correspond en pratique au *thread* d'exécution du programme. D’ailleurs, un objet de portée globale (**`ObjetApplication`**) est créé. C'est cet objet qui sera *responsable* de la création de la/des fenêtre(s) de l'application Windows. Par ailleurs, une autre classe est écrite : la classe **`CMaFenetre`** dérivée de la classe **`CFrameWnd`**. Cette classe représente la fenêtre cadre (fenêtre principale qui comporte une barre de titre) de l’application.

Exécution de ce programme.

Il est clair que le fichier source donné ci-dessus ne fait pas apparaître la fonction `WinMain()` qui est normalement le point d'entrée d'une application Windows. En fait, la fonction `WinMain()` a une implémentation *fixe* définie pour tous les programmes utilisant les MFC. La fonction `WinMain()` appelle la fonction `AfxWinMain()` (donnée ci-dessous) définie dans le fichier `WinMain.cpp`, fichier connu du compilateur et présent dans un répertoire de Microsoft Developper Studio.

```
int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPTSTR lpCmdLine, int nCmdShow)
{
        ASSERT(hPrevInstance == NULL);

        int nReturnCode = -1;
        CWinThread* pThread = AfxGetThread(); // adresse de l'application (en tant que CWinThread)
        CWinApp* pApp = AfxGetApp();          // adresse de l'application (en tant que CWinApp)

        // initialisation interne
        if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
                goto InitFailure;

        // App global initializations (rare)
        if (pApp != NULL && !pApp->InitApplication())
                goto InitFailure;

        // initialisation spécifique
        if (!pThread->InitInstance())
        {
                if (pThread->m_pMainWnd != NULL)
                {
                        TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
                        pThread->m_pMainWnd->DestroyWindow();
                }
                nReturnCode = pThread->ExitInstance();
                goto InitFailure;
        }

        nReturnCode = pThread->Run();         // lance l'exécution de la boucle des messages
                                              // méthode bloquante jusqu'au message WM_QUIT
InitFailure:
        AfxWinTerm();
        return nReturnCode;
}
```

L'objet application (`ObjetApplication`) est de portée globale, donc créé avant la fonction `AfxWinMain`(). Dans la hiérarchie des classes MFC, la class `CWinApp` (classe d'application) dérive de `CWinThread`. Autrement dit, l'objet application est un thread Windows. Exprimé différemment, toute application MFC Windows contient au moins un thread représenté par l'objet application.

On voit dans la fonction `AfxWinMain()` que les fonctions globales `AfxGetThread()` et `AfxGetApp()` récupèrent l'adresse de l'objet application (`ObjetApplication`); une fois en tant qu'objet de type `CWinThread`, une autre fois en tant qu'objet `CWinApp`, dans les deux cas c'est la même adresse numérique. Ce

procédé permet de distinguer les appels des méthodes de la classe de base `CWinThread` de ceux de la classe dérivée `CWinApp`. C'est dans cette fonction `AfxWinMain()`, qui est la même pour tous les projets MFC, que la méthode **`InitInstance()`** de l'objet application est appelée (puisque cette méthode est déclarée virtuelle dans la classe `CWinThread`, c'est celle de la classe d'application spécialisée qui est appelée dynamiquement).

En résumé, dans un projet MFC, c'est la méthode **`InitInstance()`** de la classe d'application **`xxxxxx::InitInstance()`** qui devient le point d'entrée. C'est donc dans cette méthode de la classe d'application que les fenêtres vont être créées. Dans notre exemple, la fenêtre cadre est créée et affichée en exécutant

```
m_pMainWnd=new CMaFenetre;          //crée l'objet fenêtre cadre
m_pMainWnd->ShowWindow(m_nCmdShow);  // affiche la fenêtre
```

Ensuite, la pompe à messages est exécutée dans la méthode **`CWinThread::Run()`.** Cette méthode est bloquante jusqu'à la fin de l'application, c'est à dire jusqu'à ce que la pompe à messages prélève le message **`WM_QUIT`** dans la file des messages. Ce message est en général envoyé quand l'utilisateur ferme la fenêtre.

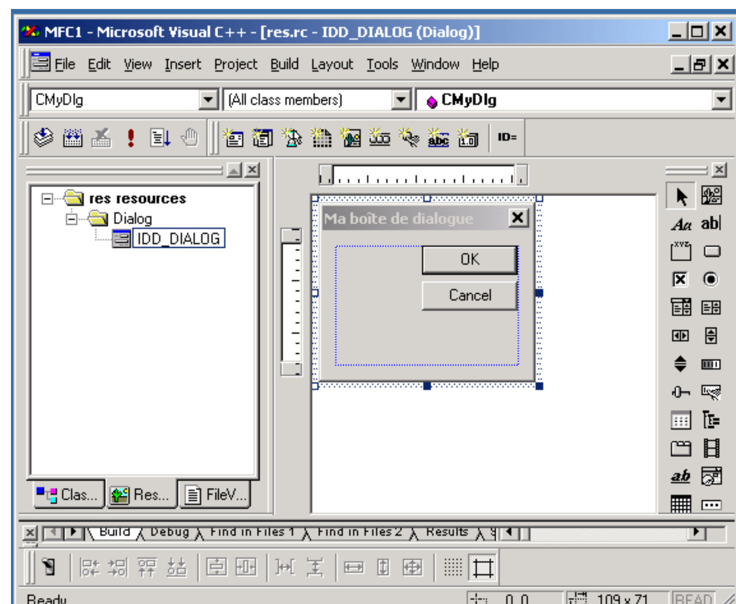
5.2 Exemple d'application dont la fenêtre principale est une boîte de dialogue

```
// App.h
#include<afxwin.h>
#include "MyDlg.h"

class CApp:public CWinApp
{
public:
        CApp(){};
        virtual ~CApp(){};

        BOOL InitInstance()
        {
                CMyDlg dlg;
                m_pMainWnd=&dlg;
                dlg.DoModal();
                return FALSE;
        };
};

CApp monApp;
```

```
// MyDlg.h
#include<afxwin.h>

class CMyDlg : public CDialog
{
public:
        CMyDlg::CMyDlg():CDialog(CMyDlg::IDD,NULL)
        {
        }

        enum { IDD = IDD_DIALOG };
};
```

L'application MFC écrite ci-dessus est une application où la fenêtre principale est une boîte de dialogue. La classe permettant de gérer la boîte de dialogue est la classe `CMyDlg`, dérivée de `CDialog`.

La méthode `CApp::InitInstance()` de l'application crée un objet automatique `dlg` de type `MyDlg`, puis appelle la méthode `dlg.DoModal()`. Cette méthode est bloquante jusqu'à ce que l'utilisateur ferme la boîte de dialogue.



Une fois fermée, la méthode `CApp::InitInstance()` retourne `FALSE` pour que l'application se termine sans lancer la boucle des messages.

Description de la boîte de dialogue

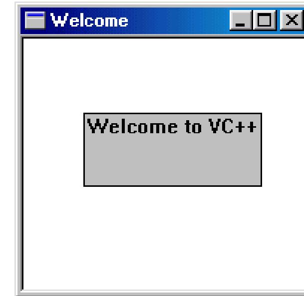
L'aspect graphique de la boîte de dialogue est décrit au moyen d'un script que l'on appelle une ressource. Ce script est écrit dans un fichier texte d'extension .rc). Ce script de

ressource contient les différentes caractéristiques de la boîte de dialogue : le titre, les contrôles, leur emplacement... La ressource est identifiée ici par la constante `IDD_DIALOG` utilisée dans le constructeur de la classe `CMyDlg`. L'IDE Visual C++ dispose d'un éditeur de ressource (Onglet Resource View) qui génère ces scripts. L'utilisateur se contente de modifier la ressource au moyen d'une interface graphique (comme sous Visual Basic ou Borland C++ Builder).

5.3 Exemple avec une fenêtre cadre et une fenêtre fille contenant un texte

Dans cet exemple, la fenêtre cadre crée un objet dynamique de la classe `CStatic`, c'est à dire une fenêtre contenant un texte.

```
#include <afxwin.h>

class CWelcomeWindow : public CFrameWnd
{
public:
                CWelcomeWindow::CWelcomeWindow()
                {
                      Create(NULL,"Welcome",WS_OVERLAPPEDWINDOW,CRect(100,100,300,300));
                      m_pGreeting=new CStatic;
                      m_pGreeting->Create("Welcome to VC++",
                                              WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
                                              CRect(40,50,160,100),this);
                }
                CWelcomeWindow::~CWelcomeWindow()
                {
                      delete m_pGreeting;
                }

private:
       CStatic * m_pGreeting;
};

class CWelcomeApp: public CWinApp
{
public:
BOOL InitInstance()
{
       m_pMainWnd=new CWelcomeWindow();
       m_pMainWnd->ShowWindow(m_nCmdShow);
       m_pMainWnd->UpdateWindow();
       return TRUE;
}

} welcomeApp;
```



En résumé ici, la fenêtre cadre de l'application est créée dans la méthode `C___App::InitInstance()` et cette fenêtre cadre crée (dans son constructeur) une fenêtre de type `CStatic`.

5.4 Exemple de classe de fenêtre cadre avec gestionnaires de messages

Dans ce dernier exemple (réalisé sans les assistants), on présente succinctement comment mettre en place des gestionnaires de message avec les MFC. L'application suivante exploite trois messages différents : le click gauche, le click droit et l'appui clavier. Les gestionnaires d'événements sont trois méthodes de la classe `CMainFrame`.

Que remarquer dans cet exemple ?

On utilise ici les macros `DECLARE_MESSAGE_MAP()`, `BEGIN_MESSAGE_MAP()` et `END_MESSAGE_MAP()` pour indiquer quels messages exploiter et comment les exploiter.

Cette application affiche un texte à l'emplacement du click et ouvre une boîte de message lors d'un appui sur le clavier.

```cpp
// MonApp.h
#include<afxwin.h>
class CMonApp : public CWinApp
{
public:
	BOOL InitInstance();
	CMonApp();
	~CMonApp();
};
```

```cpp
// MonApp.cpp
#include "MonApp.h"
#include "MainFrame.h"

CMonApp::CMonApp()
{
}

CMonApp::~CMonApp()
{
}

BOOL CMonApp::InitInstance()
{
	CMainFrame * pF=new CMainFrame;
	this->m_pMainWnd=pF;
	pF->ShowWindow(this->m_nCmdShow);

	return true;
}
CMonApp App;
```

```cpp
// MainFrame.h
#include<afxwin.h>

class CMainFrame :public CFrameWnd
{
public:
CMainFrame();
~CMainFrame();

afx_msg void OnLButtonDown(UINT,CPoint);
afx_msg void OnRButtonDown(UINT,CPoint);
afx_msg void OnKeyDown(UINT,UINT,UINT);

	DECLARE_MESSAGE_MAP()
};
```

```cpp
// MainFrame.cpp

#include "MainFrame.h"

CMainFrame::CMainFrame(){
	this->Create(NULL,"Gestion Message");
}

CMainFrame::~CMainFrame(){}

void CMainFrame::OnLButtonDown(UINT nFlags,CPoint point)
{
	this->GetDC()->TextOut(point.x,point.y,"Click Gauche");
}

void CMainFrame::OnRButtonDown(UINT nFlags,CPoint point)
{
	this->GetDC()->TextOut(point.x,point.y,"Click Droit");
}

void CMainFrame::OnKeyDown(UINT nChar,UINT nRepCnt,UINT nFlags)
{
	this->MessageBox("Appui Clavier");
}

BEGIN_MESSAGE_MAP(CMainFrame,CFrameWnd)

	ON_WM_LBUTTONDOWN()
	ON_WM_RBUTTONDOWN()
	ON_WM_KEYDOWN()

END_MESSAGE_MAP()
```

5.5 Les macros de gestion de la table des messages

La table des messages (MESSAGE_MAP) est un tableau statique attaché à une classe qui traite des messages (il y a une table des messages par classe). Ce tableau contient la liste des messages que la classe souhaite intercepter et les adresses des gestionnaires (méthodes de la classe) adéquats. La gestion de la table des messages repose sur l'utilisation de macros.

Le développement de la macro **DECLARE_MESSAGE_MAP()** (définie dans AfxWin.h) conduit à compléter la classe avec les déclarations suivantes :

```
private:
      static const AFX_MSGMAP_ENTRY _messageEntries[]; //table des messages
protected:
      static AFX_DATA const AFX_MSGMAP messageMap;
      static const AFX_MSGMAP * PASCAL _GetBaseMessageMap();
      virtual const AFX_MSGMAP * GetMessageMap() const;
```

Le développement des macros dans le fichier d'implémentation donne :

// dévelopement de **BEGIN_MESSAGE_MAP(theClass,baseClass)**

```
const AFX_MSGMAP * PASCAL theClass::_GetBaseMessageMap()
{
      return &baseClass::messageMap;
}

const AFX_MSGMAP * theClass::GetMessageMap() const;
{
      return &theClass::messageMap;
}

AFX_DATADEF const AFX_MSGMAP theClass::messageMap=
{
      &theClass::_GetBaseMessageMap,
      &theClass::_messageEntries[0]
}

// la table des messages est un tableau statique contenant des 6-uplets
const AFX_MSGMAP_ENTRY theClass::_messageEntries[]=
{
```

// développement de la macro **ON_WM_KEYDOWN()**

```
{ WM_KEYDOWN, 0, 0, 0, AfxSig_vwww,
(AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, UINT,UINT))&OnKeyDown },
```

// développement de la macro **END_MESSAGE_MAP()**

```
{0,0,0,0,AfxSig_end,(AFX_PMSG)0}\
};
```

Entre **BEGIN_MESSAGE_MAP(,)** et **END_MESSAGE_MAP()**, chaque macro (**ON_WM_KEYDOWN, ON_WM_LBUTTONDOWN, ON_WM_RBUTTONDOWN)** enrichit le tableau statique **_messageEntries[]** avec un 6-uplet contenant le numéro de message et l'adresse de la méthode (gestionnaire de message) qui doit traiter le message. En résumé, les macros de gestion de la table des messages :

. créent un tableau statique **theClass::_messageEntries[]**
. créent des méthodes d'accès à ce tableau : **virtual theClass::GetMessageMap()**
. remplissent le tableau avec des 6-uplets contenant chacun un numéro de message et l'adresse du gestionnaire pour ce message. Cette table est utilisée pour exécuter un gestionnaire en réponse à un message prélevé par la pompe à messages.

5.6 Qu'avons-nous appris jusqu'ici ?

Dans ces premiers exemples utilisant les MFC, nous avons vu :

- l'objet application (objet global de la classe d'application) représente le thread d'exécution. Ce thread contient notamment la pompe à messages qui prélève les messages dans la file des messages.
- une fenêtre est gérée à travers un objet de type `CWnd` de la bibliothèque MFC

De manière générale, une fenêtre est gérée à travers un objet de type `CWnd` (classe de base des fenêtres). Mais cette classe est spécialisée (dérivée) en différentes autres classes de fenêtres plus spécifiques. A titre d'exemple, les classes suivantes dérivent (directement ou indirectement) de la classe `CWnd`

`CStatic` : fenêtre qui contient une zone d'affichage de texte. Cette classe contient donc des méthodes spécifiques pour gérer le texte

`CButton:` un bouton est aussi une fenêtre Windows avec des caractéristiques particulières.

`CDialog` : les objets de cette classe permettent la gestion des boîtes de dialogue

Il faut néanmoins faire une « distinction » entre l'objet de type `CWnd` et la fenêtre Windows qu'il permet de gérer. Dans certains cas, un objet de type `CWnd` (ou dérivé) peut exister sans que la fenêtre Windows qu'il doit gérer n'ait encore été créée. Cela arrive quand l'objet `CWnd` est instancié mais que la fonction `Create` de l'API n'a pas encore été appelée.

Exemple :

```
CDialog dlg;        // l'objet boîte de dialogue est instancié,
                    // mais la fenêtre Windows n'est pas encore créée

dlg.Create(arg0,arg1,...);       // la fenêtre est créée ici, c-a-d enregistrée dans
                                 // Windows, mais elle n'est pas encore visible

dlg.Show(SW_SHOW);  // la boîte de dialogue est désormais visible.
```

Bien que beaucoup plus concise que la programmation Windows en C, la programmation MFC sous cette forme reste assez délicate. C'est pourquoi, Visual C++ met à disposition des assistants pour générer automatiquement du code source C++ : il s'agit des assistants `AppWizard`, `ClassWizard`.

L'environnement Visual C++ met à disposition des assistants pour faciliter la programmation d'une application. Lorsqu'on utilise ces assistants, la programmation se fait en deux étapes. Dans un premier temps, l'assistant `AppWizard` génére le canevas de différentes classes impliquées dans l'application, notamment de la classe dérivée de `CWinApp` et de celle dérivée de la fenêtre principale : soit `CFrameWnd,` soit `CDialog` si l'application est de type "**Dialog Based**". Ensuite, ce squelette peut être complété/modifié au moyen de l'assistant `ClassWizard`. Ce dernier assistant va faciliter la mise en place de gestionnaires de messages et l'association de classes MFC à des ressources. On rappelle que les ressources sont des scripts permettant de décrire certaines caractéristiques des fenêtres ou des contrôles utilisés.

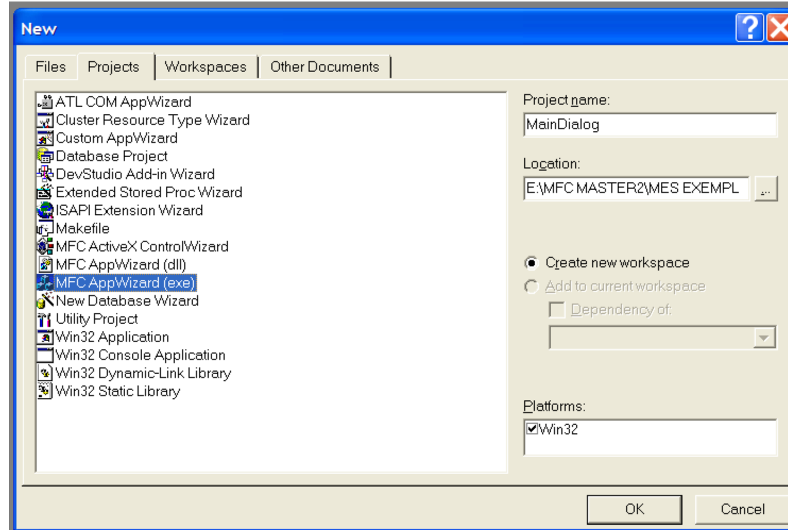
Dans un premier temps, on va se concentrer sur le développement d'applications de type « Boîte de Dialogue », ce qui permet de se familiariser avec les assistants sans risquer de « se perdre dans les classes ». Toutes les techniques de développement (gestionnaires de messages, gestion des contrôles, utilisation de boîtes de dialogue) vues ici seront directement réutilisables dans des projets plus complexes de type SDI (Single Document Interface) ou MDI (Multi Document Interface).

6.1 Application « Dialog Based » réalisée avec l'assistant AppWizard

On va utiliser les assistants pour réaliser une première application dont la fenêtre principale sera une boîte de dialogue. Il ne faut pas utiliser de nom farfelu pour nommer le projet car l'assistant **`AppWizard`** utilise ce nom pour nommer des classes ! Le rôle de l'assistant `AppWizard` est de créer le « squelette de l'application ».

Lancement de l'assistant `AppWizard` : sélectionner File | New | Projects | **MFC AppWizard (.exe)**

Project Name = nom du projet (ici `MainDialog`)
Location : chemin du répertoire qui contiendra le projet



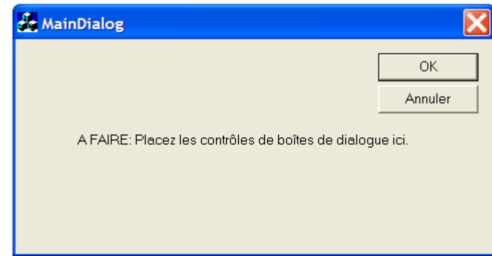
Step 1 : sélectionner « **Dialog Based** »

Step 2/4 : garder les options cochées par défaut (**About box, 3D controls, ActiveX Controls**)

Step 3/4 : les assistants peuvent génèrer des commentaires (préférable de garder cette option). Le code des classes MFC est soit utilisé dans des DLL (Dynamically Linked Library) partagées (`shared DLL`) ou ajouté dès le linkage à partir d'une librairie (`statically linked`). Dans le premier cas, l'application utilise implicitement des DLL du répertoire système (ces DLL ont été installées lors de l'installation de Visual C++). Dans le second cas, elle est autonome puisque tout le code des MFC est ajouté au code exécutable, en contrepartie le fichier exécutable est plus volumineux.

Step 4/4 : résume les classes générées (cliquer sur **Finish**)

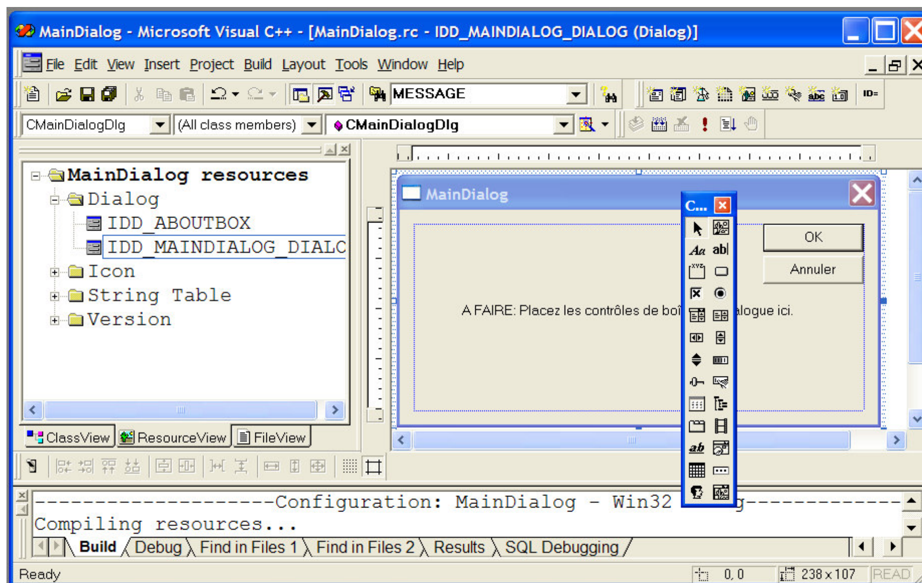
Compiler/Linker : lancer la commande [Build | Build (F7)] pour produire l'exécutable de ce projet puis l'exécuter (CTRL+F5). On obtient l'application ci-contre.



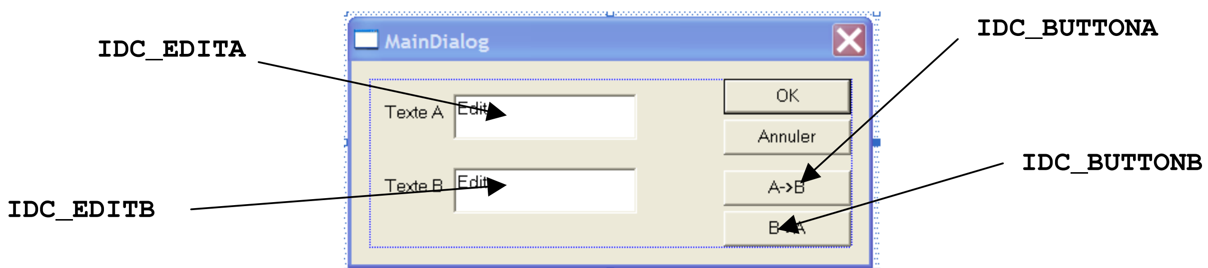
Le rôle du programmeur est de faire évoluer ce squelette initial : en complétant les méthodes générées, en en ajoutant de nouvelles, en ajoutant des classes etc. Un autre assistant nommé **`ClassWizard`** va nous permettre d'ajouter des gestionnaires de messages, de créer des classes associées à des ressources de boîte de dialogue, d'associer des objets à des contrôles etc.

6.2 Modifier la ressource de boîte de dialogue principale

L'environnement Visual C++ dispose de différents types d'éditeurs, il y en a notamment un pour les ressources. Les ressources sont des scripts (on peut aussi les ouvrir comme des fichiers texte) décrivant les menus, les dimensions des boîtes de dialogue, les emplacements et la taille des contrôles dans les boîtes de dialogue, etc. L'éditeur de ressource permet, via une interface graphique, de modifier le script de ressource.



Ajouter deux contrôles `Button`, deux zones d'édition de texte, deux textes statiques à la ressource [**`Resource View | Dialog | IDD_MAINDIALOG_DIALOG]`**. Pour chaque contrôle, modifier l'ID mis par défaut par l'éditeur ([click droit sur le contrôle |Properties]). Ces identifiants seront utilisés par la suite pour référencer les contrôles : **`IDC_BUTTONA et IDC_BUTTONB` pour les boutons, `IDC_EDITA et IDC_EDITB` pour les zones d'édition.**



MCours.com

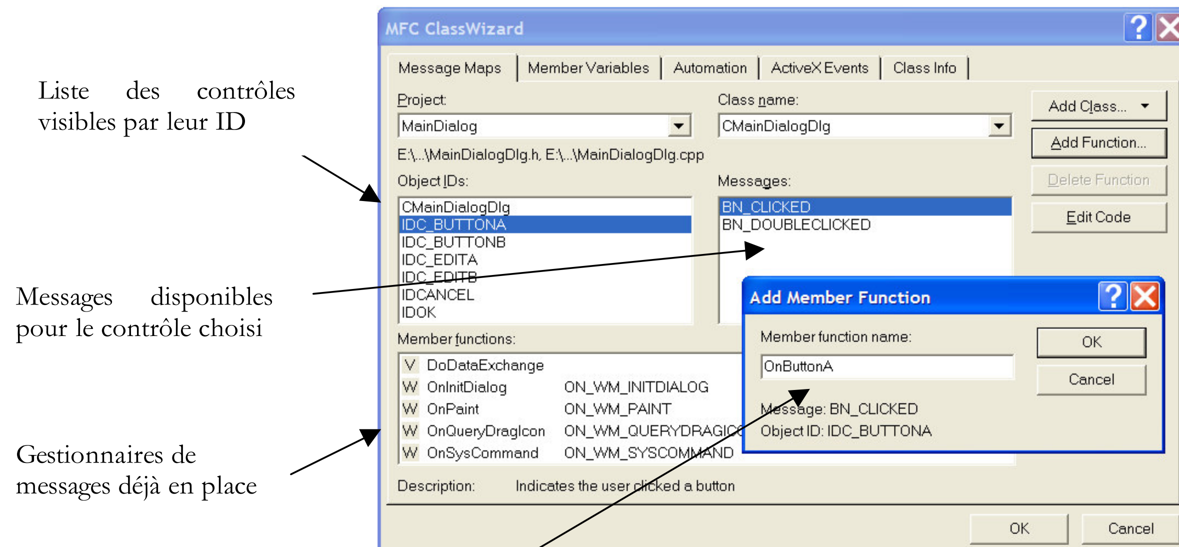
6.3 Ajouter des gestionnaires de message à l'aide de ClassWizard

On souhaite simplement que le click sur le bouton **`A->B`** copie le contenu de la zone A dans la zone B, l'inverse pour l'autre bouton. Il faut pour cela mettre en place un gestionnaire de click par bouton. C'est **`ClassWizard`** qui va réaliser ces opérations.

Lancer `ClassWizard` (accessible depuis le menu contextuel). Choisir l'onglet **`Message Maps`** :

> **Class name** : la classe dans laquelle placer le gestionnaire de messages : **`CMainDialogDlg`**
> **Object Ids** : choix du contrôle auquel associer un gestionnaire : **`IDC_BUTTONA/IDC_BUTTONB`**
> **Messages** : choix du message à traiter : **`BN_CLICKED`**

Lorsque tout est sélectionné, cliquer sur **`Add Function`**. L'assistant propose un nom (constitué à partir de l'ID du contrôle) pour le gestionnaire de message : ici **`OnButtona()`** (ce nom est modifiable).



L'assistant a complété la classe **`CMainDialogDlg`** de la manière suivante .

Il a ajouté deux méthodes `OnButtonA()` et `OnButtonB()`.

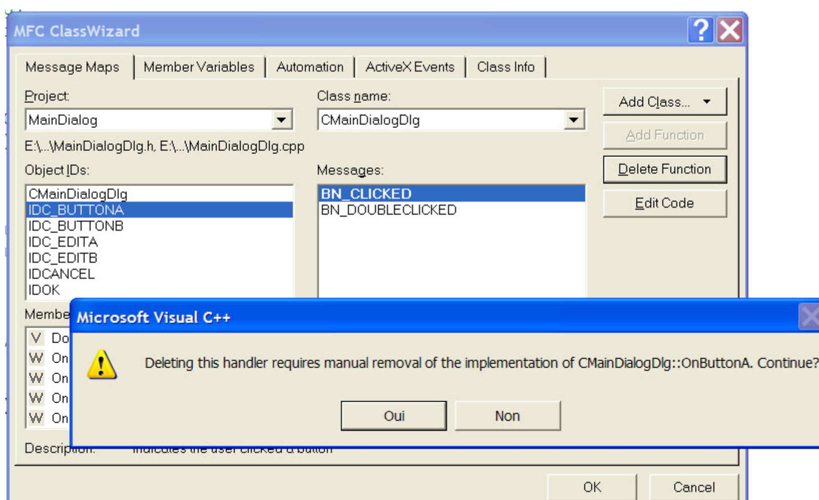
```
-----------------------------------------------------------------------------
// MainDialogDlg.h (vue partielle)

class CMainDialogDlg : public CDialog
{
        //…

        // Generated message map functions
        //{{AFX_MSG(CMainDialogDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
        afx_msg HCURSOR OnQueryDragIcon();
        afx_msg void OnButtonA(); // prototypes ajoutés par ClassWizard
        afx_msg void OnButtonB();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

-----------------------------------------------------------------------------
```

ClassWizard a défini ces méthodes dans le fichier **MainDialogDlg.cpp** et a complété la table des messages au moyen de deux appels de macros.

```
----------------------------------------------------------------------------
// MainDialogDlg.cpp (vue partielle)

        //…

BEGIN_MESSAGE_MAP(CMainDialogDlg, CDialog)
        //{{AFX_MSG_MAP(CMainDialogDlg)
        ON_WM_SYSCOMMAND()
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
        ON_BN_CLICKED(IDC_BUTTONA, OnButtonA)   //ajout d'une entrée à la table des messages
        ON_BN_CLICKED(IDC_BUTTONB, OnButtonB)  //ajout d'une entrée à la table des messages
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

        //…

// gestionnaire de click sur le bouton IDC_BUTTONA
void CMainDialogDlg::OnButtonA()
{
        // TODO: Add your control notification handler code here

        // Toujours compléter le gestionnaire
        // JUSTE APRES le commentaire TODO

        MessageBox("Click sur bouton A->B");
}

// gestionnaire de click sur le bouton IDC_BUTTONB
void CMainDialogDlg::OnButtonB()
{
        // TODO: Add your handler code here

        MessageBox("Click sur bouton B->A");
}
----------------------------------------------------------------------------
```

6.4 Supprimer un gestionnaire de message à l'aide de ClassWizard

L'assistant `ClassWizard` permet aussi la suppression d'un gestionnaire de message. Il suffit de le sélectionner dans la liste des messages traités (ils apparaissent en gras) et de cliquer sur **Delete Function**. `ClassWizard` retire alors le prototype dans la définition de la classe (dans le fichier header) et supprime la macro **ON_xxx(,)** de la liste des macros de gestion de la table des messages. En revanche, comme l'indique le message ci-dessous, il faut supprimer la définition de la méthode « à la main » dans le fichier source d'implémentation de la classe.



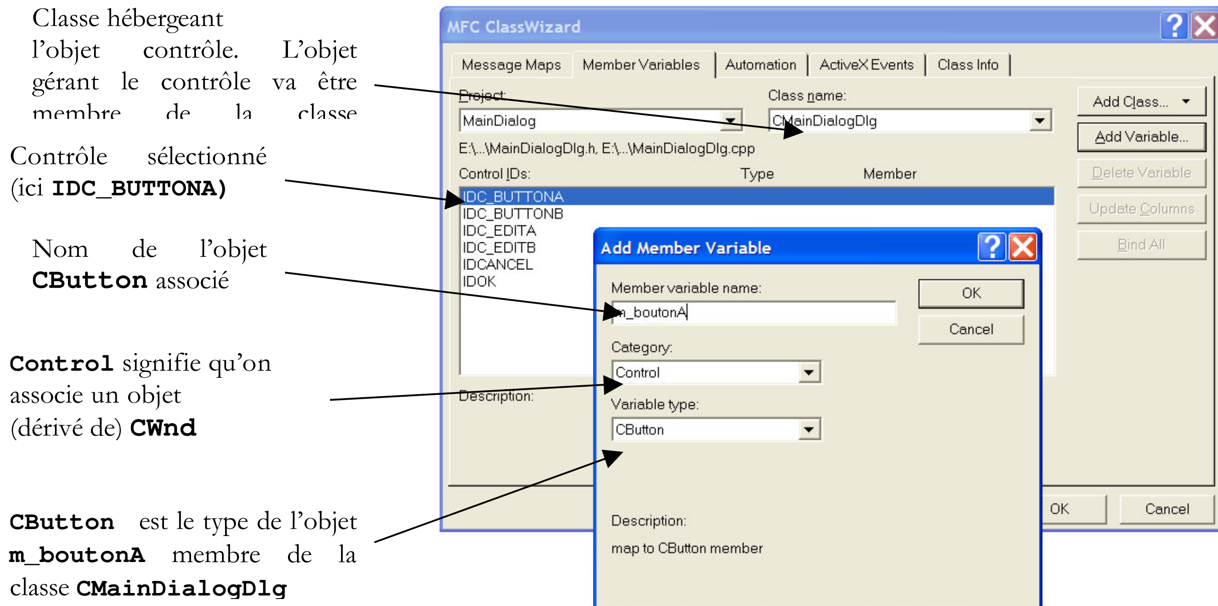
6.5 Exploiter les contrôles dans une boîte de dialogue

Là encore, ClassWizard va nous permettre d'associer des objets (ou des variables primitives, selon le cas) aux contrôles disposés dans la ressource. On va exploiter les contrôles « via **»** ou « grâce à » ces objets associés.

6.5.1 Associer un objet de type `CWnd` à un contrôle avec ClassWizard

On va associer deux objets de la classe **`CButton`** aux contrôles bouton et deux objets de la classe **`CEdit`** aux zones d'édition. Ces quatre objets vont apparaître comme des **objets membres** de la classe de boîte de dialogue (c'est-à-dire **`CMainDialogDlg`** dans notre cas).

Lancer ClassWizard et sélectionner l'onglet **`Member Variables.`** Sélectionner l'ID d'un contrôle puis cliquer sur **`Add Variable.`**



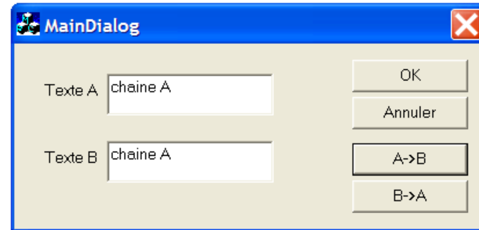
Faire la même manipulation pour associer un objet **`CMainDialogDlg::m_boutonB`** au contrôle **`IDC_BUTTONB`**, et deux objets **`CMainDialogDlg ::m_editA`** et **`CMainDialogDlg ::m_editB`** de type **`CEdit`** aux contrôles **`IDC_EDITA`** et **`IDC_EDITB`**.

En quittant ClassWizard, la classe **`CMainDialogDlg`** a été modifiée de la façon suivante :

```
-----------------------------------------------------------------
// class MainDialogDlg.h (vue partielle)

class CMainDialogDlg : public CDialog
{
	//…
public:
	CMainDialogDlg(CWnd* pParent = NULL); // standard constructor

	// Dialog Data
	//{{AFX_DATA(CMainDialogDlg) // bannières qu'il ne faut pas modifier !
	enum { IDD = IDD_MAINDIALOG_DIALOG};
	CEdit    m_editB;    // les objets membres créés par ClassWizard
	CEdit    m_editA;
	CButton  m_boutonB;
	CButton  m_boutonA;
	//}}AFX_DATA              // bannières qu'il ne faut pas modifier !

};
-------------------------------------------------------------------
```

ClassWizard a aussi modifié la méthode **`CMainDialogDlg::DoDataExchange()`** (cette méthode relie chaque ID de contrôle à un objet de gestion du contrôle). On peut dès lors compléter les gestionnaires de click sur les boutons de façon à obtenir le comportement souhaité.

```
-----------------------------------------------------------------
// class MainDialogDlg.cpp (vue partielle)

	//…
void CMainDialogDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CMainDialogDlg)
	DDX_Control(pDX, IDC_EDITB, m_editB);
	DDX_Control(pDX, IDC_EDITA, m_editA);
	DDX_Control(pDX, IDC_BUTTONB, m_boutonB);
	DDX_Control(pDX, IDC_BUTTONA, m_boutonA);
	//}}AFX_DATA_MAP
}
	//…
```

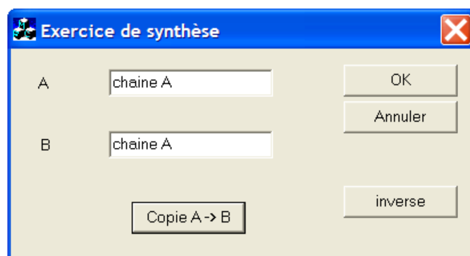


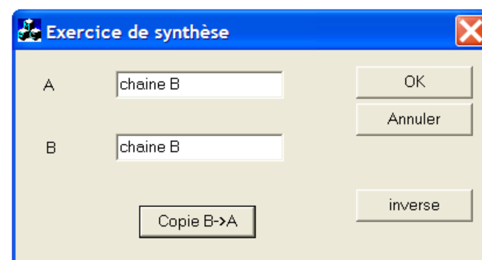

```
void CMainDialogDlg::OnButtonA()
{
	// TODO: Add your control notification handler code here
	CString str; // CString est la classe MFC de gestion des chaînes
	m_editA.GetWindowText(str); //str est une sortie de CEdit::GetWindowText(CString &)
	m_editB.SetWindowText(str); //str est une entrée de CEdit::SetWindowText(LPCSTR)
}

void CMainDialogDlg::OnButtonB()
{
	// TODO: Add your control notification handler code here
	CString str;
	m_editB.GetWindowText(str);
	m_editA.SetWindowText(str);
}
 -----------------------------------------------------------------
```

6.5.2 Exercice de synthèse : création du projet, modification des ressources, ajout de gestionnaires de messages, exploitation des contrôles

Pour synthétiser tout ce que l'on vient de voir, refaire un projet complet de type **« Dialog Based »** tel que la boîte de dialogue principale (voir les copies d'écran ci-dessous) ait le comportement suivant :

- le bouton **`inverse`** permet de changer le sens de la copie (et le texte du bouton de copie)
- le bouton **`Copie ->`** réalise la copie d'une zone d'édition vers l'autre
- le sens de la copie peut être géré grâce à membre privé **`BOOL CMainDialogDlg::m_sensCopie`**
- ce membre doit être **initialisé**, soit dans le constructeur **`CMainDialogDlg::CMainDialogDlg(CWnd *)`**, soit dans le gestionnaire **`BOOL CMainDialogDlg::OnInitDialog()`**, ce gestionnaire est appelé juste avant que la boîte de dialogue s'affiche.



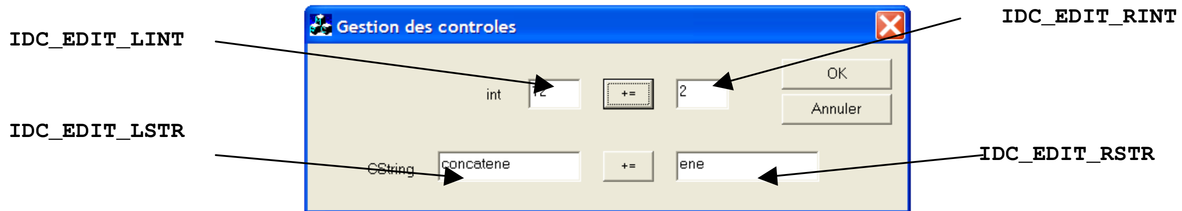


Remarque : faire ce projet à partir du projet précédent serait également un bon exercice (vérifiez dans ce cas que vous êtes capable de modifier le nom des gestionnaires de messages, les objets associés aux contrôles, le titre de la boîte de dialogue…

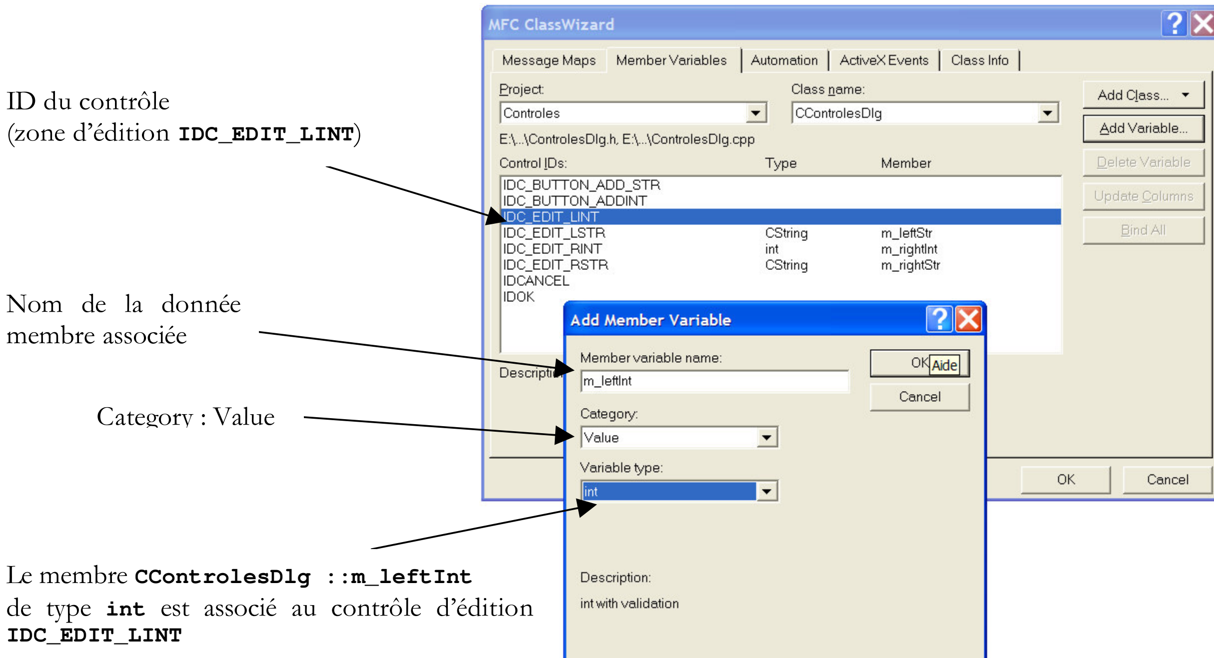
6.5.3 Associer une variable à un contrôle avec ClassWizard

Dans ce qui précédait, on associait des objets dérivés de classes de fenêtre (**`CButton`** et **`CEdit`** dérivent de **`CWnd`**) aux contrôles de la boîte de dialogue. ClassWizard permet d'associer d'autres sortes de variable aux contrôles. Pour les zones d'édition, on peut par exemple associer des objets **`CString`** (objets chaîne de caractère) voire même des variables de types primitifs `int, float, UINT (unsigned int), BYTE (octet non signé)`

Nous allons réaliser l'application ci-dessous en utilisant cette technique. Les deux boutons permettent de réaliser l'opération zone d'édition de gauche += zone d'édition de droite. Dans le premier cas, les zones d'édition gèrent des entiers, dans le second des chaînes de caractère **`CString`** (+= représente alors une concaténation).



Lancer ClassWizard puis aller dans l'onglet **`Member Variables`**., sélectionner un contrôle par son ID puis cliquer sur **`Add Variable`**.



Après avoir associé deux données membres de type **`int`** aux zones d'édition **`IDC_EDIT_LINT`** et **`IDC_EDIT_RINT`**, et deux données membre de type **`CString`** aux autres zones d'édition, la classe a été modifiée de la manière suivante.

```
-----------------------------------------------------------------
// ControlesDlg.h (vue partielle)

class CControlesDlg : public CDialog
{
      //{{AFX_DATA(CControlesDlg)
      enum { IDD = IDD_CONTROLES_DIALOG };
      int           m_rightInt;  // associé à IDC_EDIT_RINT
      CString       m_leftStr;   // associé à IDC_EDIT_LSTR
      CString       m_rightStr;  // associé à IDC_EDIT_RSTR
      int           m_leftInt;   // associé à IDC_EDIT_LINT
      //}}AFX_DATA
};
-----------------------------------------------------------------
```

```
---------------------------------------------------------------
// ControlesDlg.cpp (vue partielle)

// constructeur de la boîte de dialogue
CControlesDlg::CControlesDlg(CWnd* pParent /*=NULL*/): CDialog(CControlesDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CControlesDlg)
		m_rightInt = 0;		// initialisation des données membres
		m_leftStr = _T("");
		m_rightStr = _T("");
		m_leftInt = 0;
	//}}AFX_DATA_INIT
	// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CControlesDlg::DoDataExchange(CDataExchange* pDX)
{
	/* cette fonction virtuelle est utilisée par la méthode
	CDialog::UpdateData(BOOL sens) pour réaliser les échanges entre les
	données membres et les fenêtres des contrôles */

	/* les copies peuvent avoir lieu dans les deux sens

		UpdataData(TRUE); ⇔ (données membres <= contrôles)
		UpdateData(FALSE);  ⇔ (données membres => contrôles) */

	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CControlesDlg)
	DDX_Text(pDX, IDC_EDIT_RINT, m_rightInt); // échanges IDC_EDIT_RINT <-> m_rightInt
	DDX_Text(pDX, IDC_EDIT_LSTR, m_leftStr);
	DDX_Text(pDX, IDC_EDIT_RSTR, m_rightStr);
	DDX_Text(pDX, IDC_EDIT_LINT, m_leftInt);
	//}}AFX_DATA_MAP
}

	//…

// gestionnaire du bouton pour la concaténation de chaînes
void CControlesDlg::OnButtonAddString()
{
	UpdateData(TRUE);   // MAJ des données membres associées aux contrôles
	m_leftStr+=m_rightStr;
	UpdateData(FALSE); // MAJ des contrôles à partir des données associées
}

void CControlesDlg::OnButtonAddInt()
{
	UpdateData(TRUE);
	m_leftInt+=m_rightInt;
	UpdateData(FALSE);
}
---------------------------------------------------------------
```

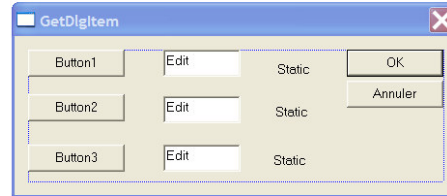
A retenir : lorsque l'on associe des variables aux contrôles (category **`Value`** dans `ClassWizard`), ce sont les appels de la méthode héritée **`CDialog::UpdatData(BOOL)`** qui activent les échanges entre les variables et les contrôles. Cette méthode utilise la méthode virtuelle **`CControlesDlg::DoDataExchange()`** surdéfinie dans notre classe spécifique (dérivée de **`CDialog)`** pour réaliser les transferts **`DDX`** (**`Dynamic Data eXchange`**) entre variables et contrôles.

```
	CDialog::UpdateData(TRUE);  //transferts DDX dans le sens contrôles->variables
	CDialog::UpdateData(FALSE); //transferts DDX dans le sens variables->contrôles
```

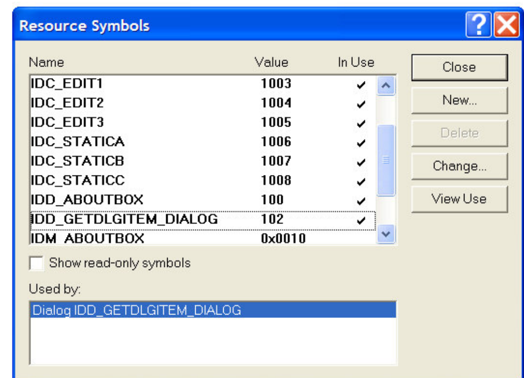
Avec cette technique, on s'affranchit néanmoins des conversions de chaînes de caractères en valeurs numériques. Ce sont les transferts DDX qui se chargent de convertir les chaînes de caractères manipulées par les contrôles en valeur numériques (`int, UINT, float…`) et vice versa.

6.5.4 Retrouver l'objet `CWnd` associé à un contrôle grâce à son identifiant (ID)

Nous avons déjà vu deux techniques différentes pour exploiter les contrôles. La troisième proposée ici permet de retrouver la fenêtre associée à un contrôle grâce à l'identifiant (ID) du contrôle. L'identifiant en question est la valeur numérique associée au contrôle lors de l'édition de la ressource. L'éditeur de ressource gère les valeurs numériques associées aux constantes symboliques **`IDxxx`**. Les valeurs numériques des `IDxxx` sont visibles dans la fenêtre **`[View | Resource Symbols]`**.



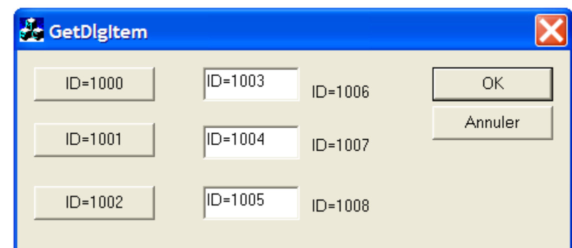
Réaliser une nouvelle application de type « **Boîte de Dialogue** » . Modifier la ressource de la boîte de dialogue en ajoutant 3 contrôles `Button`, 3 contrôle `Edit`, et 3 contrôles `Static` (modifier explicitement les ID des contrôles de texte statique).



Lancer la commande **`[View | Resource Symbols]`** pour afficher la liste des ID et leurs valeurs numériques. Ci-contre, on voit que les ID associés aux contrôles de la boîte sont les entiers allant de 1000 (pour **`IDC_BUTTON1`**) à 1008 (pour **`IDC_STATICC`**). On peut donc imaginer « parcourir ces contrôles » pour réaliser un même traitement (ce qui est une application du polymorphisme puisque ici tous les contrôles ne sont pas exactement du même type, mais ils dérivent tous de **`CWnd`**).

Remarque : il est possible de modifier les valeurs numériques des ID « à la main », tant que chaque ID est unique.

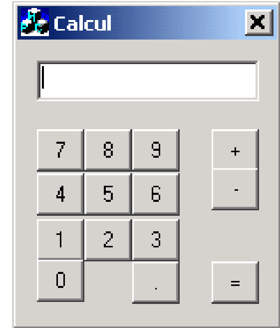
Dans la classe de boîte de dialogue, c'est la méthode **`CWnd * CDialog ::GetDlgItem(int nID)`** qui retourne l'adresse de la fenêtre associée au contrôle dont l'ID est **`nID`**.



Une application simple de cette fonctioonnalité est de parcourir tous les contrôles de la plage d'ID 1000 à 1008 (c'est-à-dire de **`IDC_BUTTON1`** à **`IDC_STATICC`**) pour leur faire afficher la valeur numérique de leur ID. Ce traitement peut être fait dans le gestionnaire `OnInitDialog()` de la boîte de dialogue.

```
// vue partielle du  gestionnaire OnInitDialog()
BOOL CGetDlgItemDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

		//… une partie du gestionnaire a été omise

	// TODO: Add extra initialization here
	CWnd * pW;   // pointeur sur objet CWnd
	CString str;
	CWnd * pW;
	CString str;
	for(unsigned i=IDC_BUTTON1;i<=IDC_STATICC; i++)
	{
		pW= GetDlgItem(i);
		str.Format("ID=%d",i);
		pW->SetWindowText(str);
	}
	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

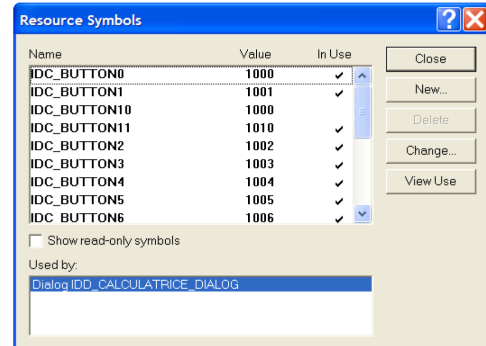
6.6 Association d'un même gestionnaire de messages à plusieurs contrôles



Dans la boîte de dialogue ci-contre, il est possible de traiter un ensemble de boutons (par exemple les boutons 0 à 9) à l'aide d'un seul gestionnaire messages (1 gestionnaire pour différents messages), ce qui peut être utile lorsque le code du gestionnaire est quasiment le même pour tous les boutons.

Les identifiants (les constantes numériques) des boutons doivent être consécutifs lors de leur définition. Tant que les identifiants restent tous différents, il est possible de modifier leurs valeurs dans le fichier **"resource.h"**. Ceci ne sera pas nécessaire si les boutons ont été créés dans l'ordre.

```
// extrait de resource.h (modifiable « à la main »)
#define IDC_BUTTON1                     1001
#define IDC_BUTTON2                     1002
#define IDC_BUTTON3                     1003
#define IDC_BUTTON4                     1004
#define IDC_BUTTON5                     1005
#define IDC_BUTTON6                     1006
#define IDC_BUTTON7                     1007
#define IDC_BUTTON8                     1008
#define IDC_BUTTON9                     1009
#define IDC_BUTTON0                      1000
#define IDC_BUTTON11                    1010
```



Ensuite, il faut ajouter un gestionnaire de messages (une méthode) pour cet ensemble de contrôles. Cet ajout ne peut pas être réalisé via ClassWizard. Ajouter une méthode (ici dans la classe de la boîte de dialogue) précédée du mot-clé **`afx_msg`**. L'argument reçu est l'ID du contrôle.

```
-------------------------------------------------
// CRangeDlg.h
class CCRangeDlg : public CDialog
{
        //…
public:
        afx_msg void OnButton(UINT nID); //declaration du gestionnaire
};
-------------------------------------------------
```

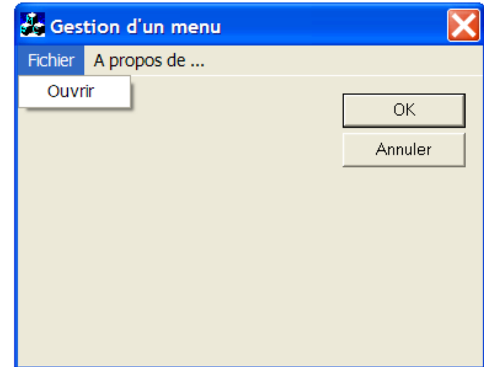
```
-------------------------------------------------
// CRangeDlg.cpp
        …
// l'implémentation de ce gestionnaire
void CCRangeDlg::OnButton(UINT nID)
{
        int numBouton= nID-IDC_BUTTON0; //numBouton est compris entre 0 et 9
        str.Format("bouton n°%d",numBouton);
        MessageBox(str);
}

BEGIN_MESSAGE_MAP(CCRangeDlg, CDialog)
        //{{AFX_MSG_MAP(CCRangeDlg)
        ON_BN_CLICKED(IDC_BPLUS, OnPlus)
        //}}AFX_MSG_MAP
        ON_CONTROL_RANGE(BN_CLICKED,IDC_BUTTON0,IDC_BUTTON9,OnButton)
END_MESSAGE_MAP()
-------------------------------------------------
```

On indique de cette façon que pour le message **`BN_CLICKED`**, pour tous les contrôles dont l'ID est compris entre **`IDC_BUTTON0`** et **`IDC_BUTTON9`**, c'est la méthode **`OnButton(UINT)`** qui est appelée.

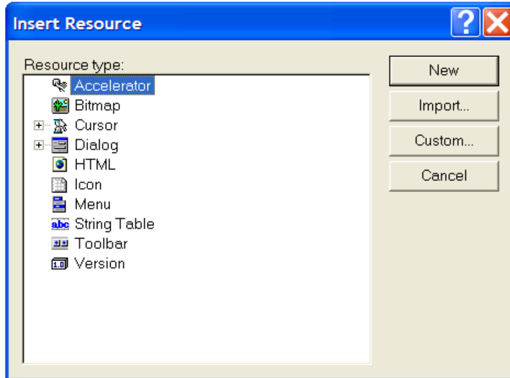
Astuce : utiliser ClassWizard pour ajouter un gestionnaire de click sur un seul bouton. ClassWizard ajoute une méthode dans la classe. Puis modifier ce qu'a fait ClassWizard conformément à ce qui précède.

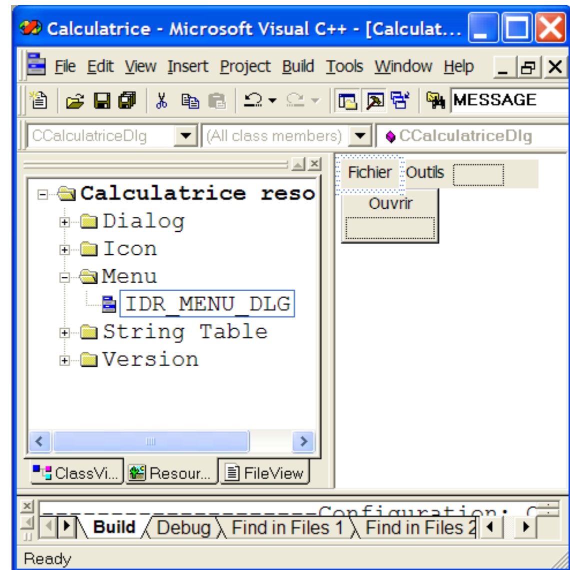
6.7 Ajout d'un menu à une boîte de dialogue

Une boîte de dialogue peut exploiter un menu (comme ci-contre).

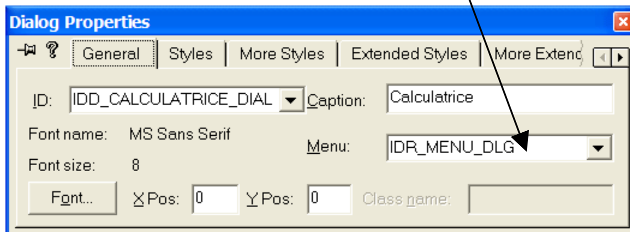


Pour cela, ajouter une ressource de menu en sélectionnant la commande **`[Insert| Resource]`** puis choisir **`Menu`**. L'éditeur de ressources permet alors de modifier le menu (penser à attribuer un ID au menu)





Ensuite, dans l'onglet **`General`** des propriétés de la boîte de dialogue, , sélectionner l'ID de la ressource de menu



Des gestionnaires d'événements peuvent alors être affectés aux différentes commandes du menu **`IDR_MENU_DLG`** via ClassWizard (comme pour les contrôles).

6.8 Les Timers

Sous Windows, il est possible de gérer des timers système qui, après installation, envoient périodiquement à l'application des messages `WM_TIMER`. Les méthodes de gestion des timers utilisées (héritées de la classe `CWnd`) sont les suivantes :

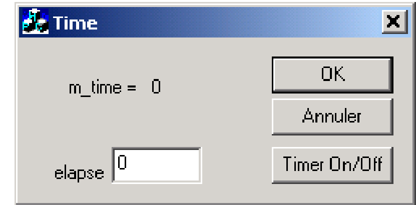
```
UINT CWnd::SetTimer(UINT_PTR nID,UINT elapse, void (CALLBACK* lpfnTimer))

BOOL CWnd::KillTimer(UINT nID)
```

`SetTimer` installe un timer d'identifiant (non nul) `nID`. Le time-out du timer est **`elapse`** exprimé en milli-secondes. Le troisième argument est l'adresse d'une fonction de traitement du message **`WM_TIMER`**. Si l'on fournit l'adresse `NULL`, le message est placé dans la queue des messages de l'application et peut être traité comme tous les messages windows, via un gestionnaire de message mis en place à l'aide de Class Wizard.

`KillTimer` désinstalle le timer d'identifiant **`nID`**.

La dernière étape est de mettre en place (à l'aide de `ClassWizard`) un gestionnaire de message sur le message **WM_TIMER**. Le message contient l'identifiant du timer qui est à l'origine de l'envoi du message. Le gestionnaire de message reçoit en argument l'identifiant du timer à l'origine du message.

```
        void CTimeDlg::OnTimer(UINT nIDEvent)
        {
                // TODO: Add your message handler code here and/or call default

                CDialog::OnTimer(nIDEvent);
        }
```



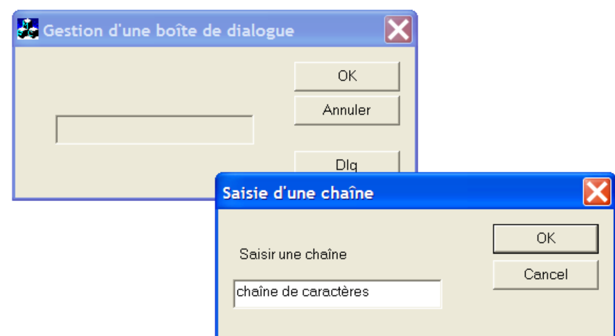
On donne ci-dessous une partie du code correspondant à l'application boîte de dialogue décrite ci-contre. Le bouton TimerOn/Off crée et supprime un timer dont le time-out dépend de l'attribut **CTimeDlg::m_elapse** associé à la zone d'édition .

A chaque message timer (`WM_TIMER`), le gestionnaire incrémente un entier non signé **CTimeDlg::m_time** et affiche cette valeur dans l'objet **CString CTimeDlg::m_timeSTR** associé au contrôle Static.

```
void CTimeDlg::OnTimerOnOff()
{
        // TODO: Add your control notification handler code here
        m_timerOn=!m_timerOn;
        UpdateData();
        if(m_timerOn) m_nTimer=SetTimer(1,m_elapse,NULL);
        else   KillTimer(m_nTimer);
}

void CTimeDlg::OnTimer(UINT nIDEvent)
{
        // TODO: Add your message handler code here and/or call default
        m_time++;
        m_timeSTR.Format("%d",m_time);
        UpdateData(FALSE);
        CDialog::OnTimer(nIDEvent);
}
```

6.9 Ajout d'une boîte de dialogue

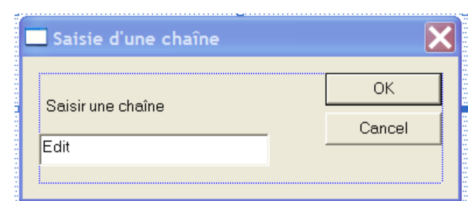
Dans toutes les applications écrites jusqu'à présent, la fenêtre principale est une boîte de dialogue qui est active aussi longtemps que l'application. On peut, en outre, ajouter des boîtes de dialogue supplémentaires pour assurer la communication avec l'utilisateur (saisie de données, affichage de données).



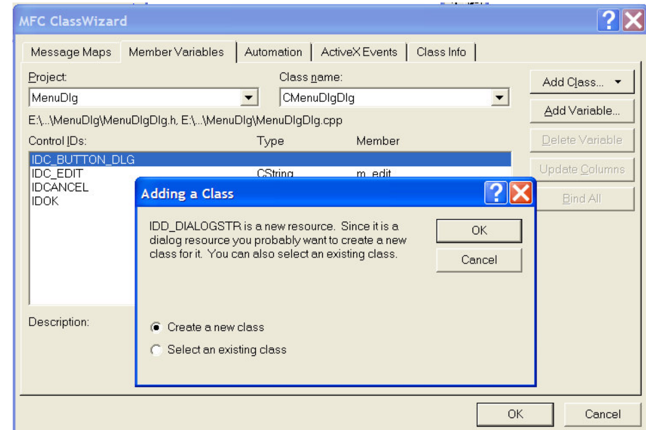
Dans l'application ci-contre, une boîte de dialogue s'ouvre sur un click sur le bouton **Dlg**. Cette boîte de dialogue permet à l'utilisateur de saisir une chaîne de caractère. Cette chaîne s'affiche dans la zone d'édition de la boîte de dialogue principale si l'utilisateur ferme la boîte de saisie en cliquant sur OK.

Tout d'abord, ajouter une ressource de boîte de dialogue.

Sélectionner la commande **[Insert| Resource]** puis choisir **Dialog**. Une nouvelle ressource de boîte de dialogue apparaît dans l'éditeur de ressources. Attribuer un identifiant (**IDD_DIALOGSTR**) et un titre à cette nouvelle boîte de dialogue.

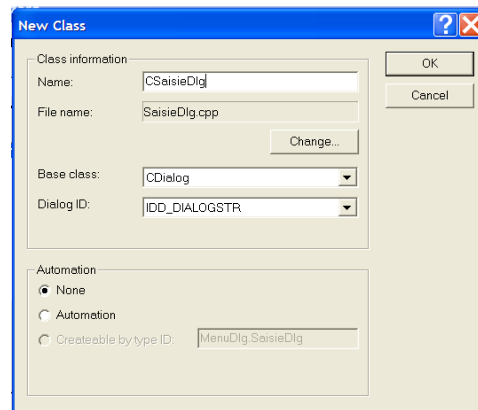


Placer des contrôles dans cette nouvelle ressource de boîte de dialogue puis lancer `ClassWizard`.



L'assistant `ClassWizard` détecte une nouvelle ressource de boîte de dialogue. Il propose don c d'associer une classe dérivée de **`CDialog`** à cette nouvelle ressource.

Laisser la sélection **`Create a new Class`** et cliquer sur OK.



Ensuite, une seconde boîte de dialogue demande un nom pour la classe de boîte de dialogue : ci-contre **`CSaisieDlg`**.

Après avoir cliqué sur OK, une nouvelle classe **`CSaisieDlg`**, dérivée de **`CDialog`**, et rattachée à la ressource **`IDD_DIALOGSTR`**, a été créée par ClassWizard et ajoutée au projet.

Avec ClassWizard, associer des variables `CString` aux zones d'édition de la boîte principale (`CMainDlg::m_edit`) et de la boîte fille (`CSaisieDlg::m_edit`).

Il ne reste plus alors qu'à créer un gestionnaire de message sur le click sur le bouton **`Dlg`**. Dans ce gestionnaire, créer un objet automatique de la classe **`CSaisieDlg`**, puis invoquer sa méthode **`DoModal()`** (voir le code ci-dessous)

```
-------------------------------------------------------------------------------
// MainDlg.Cpp (vue partielle du fichier d'implémentation de la boîte principale)

#include "SaisieDlg.h" // insertion du fichier header de la boîte de dialogue

      //… implémentation de CMainDlg

// gestionnaire de click sur le bouton Dlg
void CMainDlg::OnButtonDlg()
{
      // TODO: Add your control notification handler code here

      CSaisieDlg dlg;     // objet automatique de la classe CSaisieDlg

      if(dlg.DoModal()==IDOK)    // DoModal() retourne IDOK si la boîte est fermée avec OK
      {
            m_edit=dlg.m_edit;  // m_edit : associé au contrôle de la boîte principale
                                // dlg.m_edit : associé au contrôle de la boîte dlg
            UpdateData(FALSE);
      }
}

-------------------------------------------------------------------------------
```

7 Les boîtes de dialogue

7.1 Les boîtes de messages

Bon nombre des dialogues avec l'utilisateur consistent à poser des questions auxquelles l'utilisateur répond par Oui, Non, Recommencer, Annuler… Les boîtes de message permettent la gestion de tels échanges.

Il y a deux façons de créer des boîtes de messages. Tout d'abord, il y a la fonction de l'API Windows (cette fonction n'est membre d'aucune classe) **AfxMessageBox**. Le prototype de cette fonction est le suivant

```
int AfxMessageBox( LPCTSTR lpszText, UINT nType = MB_OK, UINT nIDHelp = 0 );
```

Les paramètres sont :

lpszText : pointe un objet **CString** ou une chaîne de caractères terminant par 0.
nType : le style de la boîte

MB_ABORTRETRYIGNORE	La boîte contient 3 boutons : **Abort**, **Retry**, et **Ignore**.
MB_OK	La boîte contient 1 bouton : **OK**.
MB_OKCANCEL	La boîte contient 2 boutons: **OK** et **Cancel**.
MB_RETRYCANCEL	La boîte contient 2 boutons: **Retry** et **Cancel**.
MB_YESNO	La boîte contient 2 boutons: **Yes** et **No**.
MB_YESNOCANCEL	La boîte contient 3 boutons: **Yes**, **No**, et **Cancel**.

Le style peut être combiné par un **OU logique** avec les styles d'icônes suivantes

MB_ICONHAND, **MB_ICONSTOP**, and **MB_ICONERROR**
MB_ICONQUESTION
MB_ICONEXCLAMATION et **MB_ICONWARNING**
MB_ICONASTERISK et **MB_ICONINFORMATION**

La valeur de retour est :

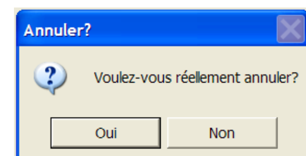
0	Si mémoire insuffisante pour afficher la boîte
IDABORT	Le bouton **Abort** a été sélectionné.
IDCANCEL	Le bouton **Cancel** a été sélectionné.
IDIGNORE	Le bouton **Ignore** a été sélectionné.
IDNO	Le bouton **No** a été sélectionné.
IDOK	Le bouton **OK** a été sélectionné.
IDRETRY	Le bouton **Retry** a été sélectionné.
IDYES	Le bouton **Yes** a été sélectionné.

La classe **CWnd** propose aussi une méthode permettant la création d'une boîte de message, son prototype est le suivant : le style **nType** prend les mêmes valeurs que ci-dessus, on peut en outre changer le titre de la boîte (second paramètre).

```
int CWnd::MessageBox( LPCTSTR lpszText, LPCTSTR lpszCaption = NULL, UINT nType = MB_OK );
```

Exemple : la boîte ci-contre est obtenue depuis une classe de fenêtre (classe dérivée de `CWnd`) sur l'appel de la méthode `CWnd::MessageBox()` suivant

```
MessageBox("Voulez-vous réellement annuler?",
                      "Annuler ?",MB_YESNO|MB_ICONQUESTION);
```



7.2 Les boîtes « Modales »

Les applications faîtes jusqu'à présent sont des applications Windows dont la fenêtre principale est une boîte de dialogue « Modale ». Une boîte de dialogue modale « garde la main » jusqu'à ce que l'utilisateur la ferme (les boîtes de messages sont aussi des fenêtres modales):

- soit en cliquant sur le bouton `OK`, `CDialog::DoMolal()` retourne alors la constante **`IDOK`**,
- soit sur le bouton `CANCEL`, `CDialog::DoModal()` retourne alors la constante **`IDCANCEL`**,
- soit sur le bouton de fermeture `X` de la fenêtre, `CDialog::DoModal()` retourne aussi **`IDCANCEL`**

On rappelle (revoir la section 6.9) que l'ajout d'une nouvelle boîte de dialogue consiste à :

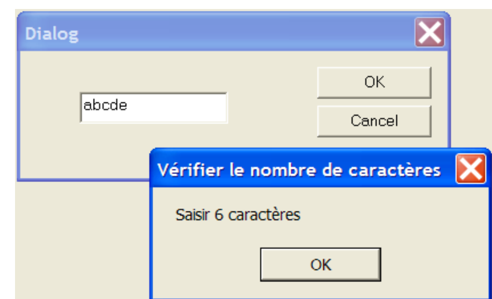
- créer une nouvelle ressource de boîte de dialogue (**`Insert | Resource`**)
- lancer `ClassWizard` qui se charge d'associer une classe dérivée de `CDialog` à cette ressource.

La classe générée par `ClassWizard` fait apparaître l'association à l'ID de la ressource (ci dessous **`IDD_DIALOG_NOUVELLE`**) au niveau du constructeur :

```
class CNouvelleDlg : public CDialog
{
	//{{AFX_DATA(CTestDlg)
	enum { IDD = IDD_DIALOG_NOUVELLE };  // IDD_DIALOG_NOUVELLE
	//}}AFX_DATA
};

CTestDlg::CTestDlg(CWnd* pParent /*=NULL*/):CDialog(CTestDlg::IDD, pParent)
{
// l'ID de la ressource est passé au constructeur de la partie héritée CDialog
}
```

Pour créer une boîte modale, il suffit alors de créer un objet automatique de la classe de boîte de dialogue puis d'appeler la méthode `CDialog::DoModal()` sur cet objet.

7.2.1 Interception des clicks sur les boutons OK/CANCEL de la boîte

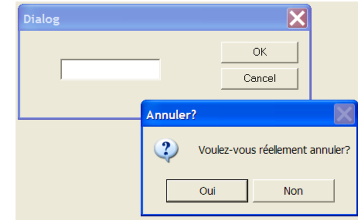


Il est parfois nécessaire de vérifier le contenu des contrôles d'une boîte de dialogue avant de la fermer. Pour faire de tels tests, successifs au choix de l'utilisateur, il faut surcharger, dans la classe de boîte de dialogue, les méthodes virtuelles **`OnOK() et OnCancel()`** (méthodes définies dans la classe `CDialog` qui ferment la boîte et retournent `IDOK/IDCANCEL`).

L'exemple ci-dessous présente le traitement réalisé pour s'assurer que le texte saisi dans la zone d'édition a exactement 6 caractères (figure ci-contre). La classe de boîte de dialogue est **`CTestDlg`**, cette classe contient un objet `CString` associé à la zone d'édition.

```
void CTestDlg::OnOK()
{
	UpdateData();		//MAJ de m_edit_str associé à la zone d'édition
	if(m_edit_str.GetLength()!=6)
	{
		MessageBox("Saisir 6 caractères",
			"Vérifier le nombre de caractères",MB_OK);
		return;		//la boîte reste "ouverte"
	}
	CDialog::OnOK();	// ferme la boîte et retourne IDOK
}
```

L'exemple ci-dessous montre comment s'assurer que l'utilisateur souhaite réellement annuler lorsqu'il clique sur le bouton `Cancel` (figure ci-contre), ou lorsqu'il ferme la boîte en cliquant sur le bouton de fermeture **X**.




```
void CTestDlg::OnCancel()
{
   if (MessageBox("Voulez-vous réellement annuler?",
                                   "Annuler ?",
                                   MB_YESNO|MB_ICONQUESTION)== IDNO )
      return; // laisse à l'utilisateur la chance de revenir sur son annulation
  CDialog::OnCancel(); // ferme la boîte et retourne IDCANCEL
}
```

7.3 Les boîtes de dialogue « Non Modales » (Modeless)

Il est important de se souvenir qu'il y a une différence entre les objets instanciés à partir des classes MFC (ou des classes dérivées) et la fenêtre que ces objets « gèrent ». Un objet instancié à partir d'une classe MFC peut exister sans que la fenêtre Windows qu'il gère soit créée. En revanche, le destructeur de l'objet détruit nécessairement la fenêtre Windows associée.

Les boîtes de dialogue non modales sont avant tout des fenêtres de boîtes de dialogue. Dans l'environnement Visual C++, pour gérer une boîte de dialogue non modale, il convient donc de créer une nouvelle ressource de boîte (`Insert | Resource [Dialog]`) puis d'associer une classe dérivée de `CDialog` à cette ressource via `ClassWizard` (la même démarche que pour les boîtes de dialogue modales).

Ensuite, il faut créer un objet de cette classe puis appeler sa méthode `CDialog::Create()`. C'est cette méthode `Create()` qui crée effectivement la fenêtre Windows associée à l'objet. Cette méthode a le prototype suivant :

```
BOOL CDialog::Create( UINT nIDTemplate, CWnd* pParentWnd = NULL )
```

Paramètres :

```
nIDTemplate : ID de la resource de boîte de dialogue
pParentWnd : adresse de la fenêtre parent. Si NULL est passé (valeur par défaut),
               la fenêtre parent est la fenêtre principale
```

Cette fois-ci, l'objet de la classe de boîte de dialogue ne peut pas être un objet automatique créé localement dans un gestionnaire de message comme c'est le cas pour les boîtes modales. Il y a plusieurs solutions.

Pour tout ce qui suit, on crée une application **`Main`** avec `AppWizard`, de type « **`Dialog Based`** », dont la classe de boîte de dialogue principale est **`CMainDlg`**. Cette boîte de dialogue principale contient un menu dont la seule entrée est une commande intitulée « **`Boîte non modale`** ». Cette commande est censée afficher une boîte non modale.

On crée en outre une ressource de boîte de dialogue pour la boîte non modale : `Insert | Resource [Dialog]`

`IDD_NONMODALE_DLG` : ID de la ressource de cette boîte.

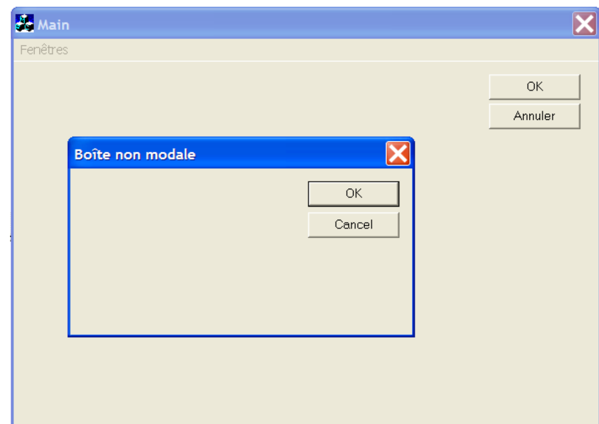
On lance `ClassWizard` pour créer la classe de boîte de dialogue associée : **`CNonModaleDlg`**

7.3.1 Ajouter un membre de type `CNonModaleDlg` à la classe de la boîte de dialogue principale

Pour que l'objet de gestion de la boîte de dialogue non modale existe suffisamment longtemps, la première solution est de mettre dans la classe de boîte de dialogue principale (**`CMainDlg`**) un objet membre de la classe **`CNonModaleDlg`**. L'objet de gestion de la boîte de dialogue non modale est ainsi créé en même temps que l'objet de gestion de la boîte de dialogue principale.

```
---------------------------------------------------------------
// MainDlg.h (vue partielle): entête de la boîte principale
      ...
#include "NonModaleDlg.h"

class CMainDlg : public CDialog
{
// Construction
public:
      CNonModaleDlg m_dlg;   // objet membre (pour la boîte non modale)

      CMainDlg(CWnd* pParent = NULL);

              ...
};
---------------------------------------------------------------
```

On peut alors appeler la méthode **`Create()`** de l'objet de boîte non modale dans le gestionnaire **`OnInitDialog()`** de la boîte principale.



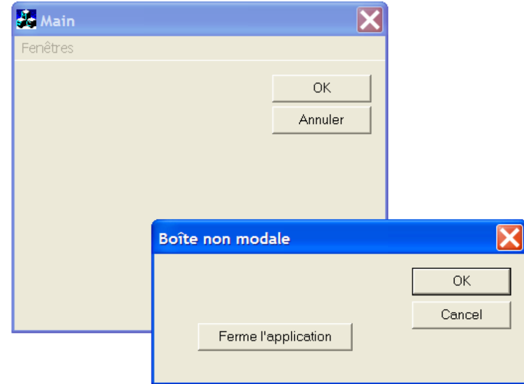

```
----------------------------------
// MainDlg.cpp (vue partielle)
      ...

BOOL CMainDlg::OnInitDialog()
{
      CDialog::OnInitDialog();
      SetIcon(m_hIcon, TRUE);
      SetIcon(m_hIcon, FALSE);

      // TODO: Add initialization here

      m_dlg.Create(IDD_NONMODALE_DLG,this);       // crée la fenêtre pour la boîte
      return TRUE;  // return TRUE  unless you set the focus to a control
}

void CMainDlg::OnBoiteNonModale()
{
      // gestionnaire de la commande "Boîte non modale" du menu de la boîte
      // principale :  affiche/cache la boîte non modale
      if(m_dlg.IsWindowVisible()) m_dlg.ShowWindow(SW_HIDE);
      else m_dlg.ShowWindow(SW_SHOW);      // rend la fenêtre visible
}
---------------------------------------------------------------
```

Remarque : selon la valeur de la propriété `Visible` de la ressource de la boîte de dialogue non modale, la boîte est visible ou non après sa création.

Echanges entre boîte principale et boîte fille

Il est possible ensuite d'ajouter des contrôles à la boîte fille (boîte non modale) et à la boîte principale. La boîte principale peut accéder à n'importe quelle partie publique de la boîte fille puisque celle-ci est un de ses objets membres. En revanche, pour que la boîte fille puisse voir sa boîte « mère » (la boîte principale), il faut qu'elle utilise la méthode `CWnd * CDialog ::GetParent()`. La navigation est alors possible dans les deux sens (moyennant un transtypage)

Exemple : fermer la fenêtre principale (et donc l'application) depuis la fenêtre fille. Il suffit pour cela d'envoyer le message **`WM_CLOSE`** à la fenêtre principale.

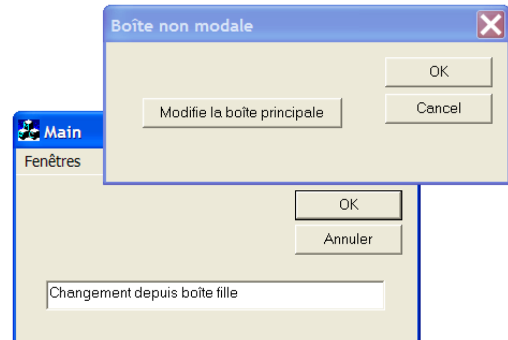


Ajouter un bouton à la boîte fille (non modale) , puis ajouter à l'aide de **`ClassWizard`**, dans la classe **`CNonModaleDlg`**, un gestionnaire de click pour ce bouton. Ce gestionnaire est le suivant :

```
--------------------------------------------
// NonModaleDlg.cpp (vue partielle)

      ...

BEGIN_MESSAGE_MAP(CNonModaleDlg, CDialog)
      //{{AFX_MSG_MAP(CNonModaleDlg)
      ON_BN_CLICKED(IDC_BUTTON_CLOSE, OnButtonClose)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CNonModaleDlg::OnButtonClose()
{
      // envoie le message WM_CLOSE à la boîte principale
      GetParent()->PostMessage(WM_CLOSE);
}
---------------------------------------------------
```

Exemple : changer un contrôle d'édition de la boîte principale depuis la boîte fille.

On ajoute à la boîte principale un contrôle d'édition et on lui associe (via `ClassWizard`) un objet de type **`CEdit`** dans la classe **`CMainDlg`**.



On donne ci-dessous le gestionnaire du click sur le bouton de la boîte fille.

```
----------------------------------
// NonModaleDlg.cpp (vue partielle)
#include "stdafx.h"
#include "Main.h"
#include "NonModaleDlg.h"
#include "MainDlg.h" // on utilise ce type dans ce fichier source
      ...
BEGIN_MESSAGE_MAP(CNonModaleDlg, CDialog)
      //{{AFX_MSG_MAP(CNonModaleDlg)
      ON_BN_CLICKED(IDC_BUTTON_MODIFIE, OnButtonModifie)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CNonModaleDlg::OnButtonModifie()
{
      // on convertit CWnd* -> CMainDlg *
      CMainDlg * pMain= static_cast<CMainDlg*>(GetParent());
      // on peut alors accéder à l'objet associé au contrôle d'édition
      pMain->m_edit_main.SetWindowText("Changement depuis boîte fille");
}
--------------------------------------------------------------------------
```

7.3.2 Créer un objet dynamique de la classe **CNonModaleDlg**

On peut aussi créer, suite à un événement ou dans la méthode **CMainDlg::OnInitDialog()**, un objet **dynamique** de la classe **CNonModaleDlg**. Il suffit pour cela de placer dans la classe **CMainDlg** un membre de type **CNonModaleDlg *** (pointeur sur la boîte fille). Il est alors important de désallouer cet objet dans le destructeur de la classe de la boîte principale (ajouter le destructeur **CMainDlg::~CMainDlg()**)

```
------------------------------------------------------------
// MainDlg.h (extrait) : classe de la boîte principale

#include "NonModaleDlg.h"

class CMainDlg : public CDialog
{
public:
	CNonModaleDlg * m_pDlg;	// pointeur sur la boîte fille
	CMainDlg(CWnd* pParent = NULL);	// standard constructor
	~CMainDlg();		// ajouter le destructeur
		...
};
------------------------------------------------------------
------------------------------------------------------------
// MainDlg.cpp (extrait) : implémentation de la classe de boîte principale

CMainDlg::CMainDlg(CWnd* pParent /*=NULL*/):CDialog(CMainDlg::IDD, pParent)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
	m_pDlg=NULL; // plus prudent
}

CMainDlg::~CMainDlg()
{
	delete m_pDlg; // désalloue l'objet de boîte
	// noter qu'il n'y a pas de problème si m_pDlg==NULL
}

BOOL CMainDlg::OnInitDialog()
{
	CDialog::OnInitDialog();
	SetIcon(m_hIcon, TRUE);
	SetIcon(m_hIcon, FALSE);

	m_pDlg=new CNonModaleDlg;	// alloue un objet
	m_pDlg->Create(IDD_NONMODALE_DLG,this);
	return TRUE;
}

void CMainDlg::OnBoiteNonModale()
{
	if(m_pDlg->IsWindowVisible()) m_pDlg->ShowWindow(SW_HIDE);
	else	m_pDlg->ShowWindow(SW_SHOW);
}
	...
------------------------------------------------------------
```

7.3.3 Détecter les fuites mémoire avec Visual C++

Lorsque l'on crée la boîte non modale, on précise que la boîte principale est la fenêtre « parent » (voir le code ci-dessous).

```
BOOL CMainDlg::OnInitDialog()
 {
        ...
        m_pDlg=new CNonModaleDlg;            // alloue un objet
        m_pDlg->Create(IDD_NONMODALE_DLG,this); //la fenêtre principale this est parent
 }
```

La conséquence est que lorsque la boîte principale est détruite, elle détruit **toutes ses fenêtres filles** aussi (elle leur envoie les messages `WM_DESTROY` et `WM_NCDESTROY`). Par conséquent, même si l'on oublie d'écrire le destructeur (rappelé ci-dessous) de la boîte principale pour désallouer l' objet pointé par **`m_pDlg`**

```
 CMainDlg::~CMainDlg(){
        delete m_pDlg; // désalloue l'objet alloué
 }
```

on verra tout de même la boîte de dialogue non modale « disparaître » en même temps que la boîte principale. Mais dans ce cas il y a une **fuite mémoire**. En debuggage, Visual C++ peut détecter cette fuite et faire un **compte rendu** sur le déroulement de l'application.

Exemple : mettre en commentaire la désallocation de l'objet dynamique dans le destructeur

```
 CMainDlg::~CMainDlg(){
        //delete m_pDlg;        // fuite mémoire voulue
 }
```

Lancer l'exécution en mode debuggage (`Build|Start Debug | Go (F5)`) et fermer l'application aussitôt (il suffit de l'exécuter et l'arrêter pour qu'il y ait une fuite). Vous verrez alors la sortie suivante dans la fenêtre **`Output`** de l'IDE Visual C++ :

```
        ...
Loaded 'C:\WINDOWS\system32\uxtheme.dll', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\MSCTF.dll', no matching symbolic information found.
Detected memory leaks!
Dumping objects ->
E:\MFC Master2\Mes Exemples\Main\MainDlg.cpp(63) : {56} client block at 0x00AA1A10,
subtype 0, 92 bytes long.
a CDialog object at $00AA1A10, 92 bytes long
Object dump complete.
The thread 0x48C has exited with code 2 (0x2).
The program 'E:\MFC Master2\Mes Exemples\Main\Debug\Main.exe' has exited with code 2
(0x2).
```

Visual C++ a détecté (**`Detected memory leaks!)`** la perte d'un objet de type **`CDialog!`** Il s'agit en fait de l'objet de la classe **`CNonModaleDlg`** alloué dynamiquement et jamais restitué au gestionnaire.

7.3.4 Créer des objets alloués dynamiquement de la classe `CNonModaleDlg` et leur assurer une « sorte » de désallocation « automatique »

Puisque toute fenêtre fille reçoît de sa fenêtre parent des messages pour la fermer, on peut intercepter ces messages pour réaliser la désallocation de mémoire.

La fenêtre fille (ici, la boîte fille non modale) reçoît d'abord le message `WM_DESTROY` puis le message `WM_NCDESTROY`. L'interception de ce second message (c'est le dernier message que la fenêtre reçoît avant d'être détruite) permet notamment la désallocation de l'objet alloué. **En fait, l'objet va s'auto-désallouer !**

Pour faire en sorte que la classe assure une désallocation automatique de l'objet alloué dynamiquement (**auto-cleanup**), il faut surdéfinir la méthode virtuelle **CDialog::PostNcDestroy()**.

Cliquer droit sur la classe **CNonModaleDlg**, puis sélectionner **Add Virtual Function**, puis sélectionner **PostNcDestroy**. Compléter cette méthode virtuelle de la façon suivante :

```
void CNonModaleDlg::PostNcDestroy()
{
	//TODO:
	delete this;		// l'objet libère lui-même la mémoire allouée
	CDialog::PostNcDestroy(); // appel de la méthode héritée.
}
```

Mises en garde : dans ce cas, on ne doit absolument pas créer d'objets automatiques de la classe **CNonModaleDlg** !

7.3.5 Comment empêcher la création d'objets automatiques d'une classe

C'est assez simple : il suffit de rendre tous les constructeurs privés, et d'ajouter une fonction membre statique qui retourne l'adresse d'un objet alloué.

```
---------------------------------------------------------
// classe dont on ne peut pas faire d'objet automatique
class AClass
{
private:
	AClass();				// non utilisable hors classe
	AClass(const Aclass & );
	void operator=(const Aclass &);
public:
	virtual ~AClass();			// destructeur accessible
	static AClass * CreateObject();	// fonction membre statique
};
---------------------------------------------------------

---------------------------------------------------------
//Aclass.cpp : implementation
AClass::AClass(){
	...
}

AClass::~AClass(){
	...
}

AClass * AClass::CreateObject()
{
	return new AClass;		// alloue un objet et renvoie son adresse
}
---------------------------------------------------------
```

Comment créer un objet dynamique ? En utilisant la fonction membre statique (on peut l'appeler sans objet) qui retourne l'adresse d'un nouvel objet alloué.

```
void main()
{
	// Aclass A ;  -> impossible car le constructeur sans argument est privé
	AClass * pC=AClass::CreateObject();
	delete pC;	// possible car le destructeur est public
}
```

La fonction membre statique **`CreateObject()`** peut être ajoutée « automatiquement » à une classe dérivée de **`CObject`** grâce aux macros **`DECLARE_DYNCREATE(cls)`** et **`IMPLEMENT_DYNCREATE(cls,basecls)`**. Puisque **`CDialog`** dérive de **`CObject`** (d'ailleurs **`CObject`** est la classe mère de presque toutes les classes MFC), on peut ajouter cette fonctionnalité à notre classe de boîte de dialogue. Ca donne ce qui suit :

```
-------------------------------------------------------
// NonModaleDlg.h (extrait)

class CNonModaleDlg : public CDialog
{
	DECLARE_DYNCREATE(CNonModaleDlg)		//déclare CreateObject()

private:
	CNonModaleDlg(CWnd* pParent = NULL);   // interdit les objets automatiques

public:

	...
};
-------------------------------------------------------

-------------------------------------------------------
// NonModaleDlg.cpp (extrait)
#include "stdafx.h"
#include "Main.h"
#include "NonModaleDlg.h"
#include "MainDlg.h"

	IMPLEMENT_DYNCREATE(CNonModaleDlg,CDialog)   // implémente CreateObject()
	...

CNonModaleDlg::CNonModaleDlg(CWnd* pParent):CDialog(CNonModaleDlg::IDD, pParent)
{
}

void CNonModaleDlg::PostNcDestroy()
{
	delete this;		// auto cleanup
	CDialog::PostNcDestroy();
}
	...
-------------------------------------------------------
-------------------------------------------------------
// MainDlg.cpp (extrait)
	...
BOOL CMainDlg::OnInitDialog()
{
	CDialog::OnInitDialog();
	SetIcon(m_hIcon, TRUE);
	SetIcon(m_hIcon, FALSE);

	m_pDlg=static_cast<CNonModaleDlg*>(CNonModaleDlg::CreateObject());
	m_pDlg->Create(IDD_NONMODALE_DLG,this);
	return TRUE;  // return TRUE  unless you set the focus to a control
}
-------------------------------------------------------
```

8.1 Introduction

Nous allons développer un exemple où un même objet fournira ses données à travers différentes vues. Une telle application repose souvent sur une architecture logicielle appelée **Modèle-Vue-Contrôleur (MVC)**.

La partie **Modèle** : contient les classes métier, c'est le cœur de l'application.

La partie **Contrôleur** : ce qui permet à l'utilisateur de saisir et de contrôler l'application (les boutons, les zones de saisies, les boutons radio, les combo box…)

La partie **Vue** : les sorties pour l'utilisateur.

Il est à noter que les fenêtres des applications Windows contiennent souvent à la fois une partie Contrôleur et une partie Vue.

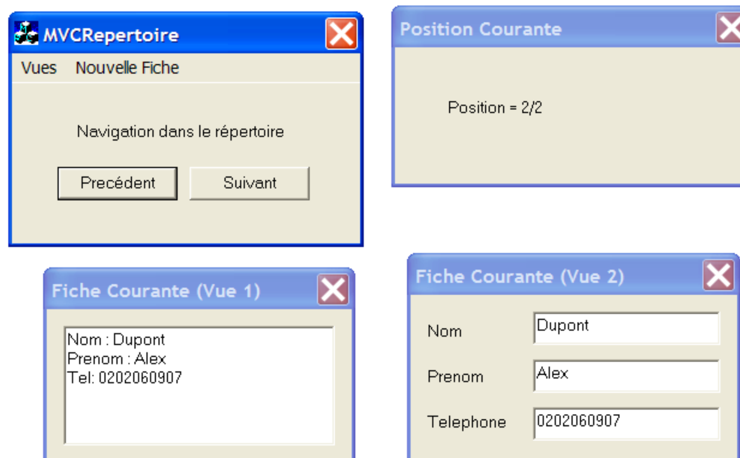
Dans le modèle MVC, l'objectif premier est de découpler la partie métier (le modèle) de la partie « Interface Utilisateur » (les contrôleurs + les vues). Par ailleurs, le même modèle peut être visualisé à travers différentes vues : un modèle peut être lié à plusieurs vues. Soit les vues sont complémentaires en fournissant chacune une partie des données, soit elles présentent les mêmes informations sous des formats différents.

L'objectif de cette section est de présenter une organisation possible des différentes classes (classe métier, classes d'interface utilisateur…) pour implémenter une application sur le modèle MVC.

Exemple traité : L'application permet à l'utilisateur de saisir et de visualiser des numéros de téléphones. Le cœur de l'application (le modèle) repose sur un objet `Repertoire` qui stocke des fiches (objets `Fiche`) contenant un nom, un prénom et un numéro de téléphone.

Il s'agit d'une application MFC où la fenêtre principale est une boîte de dialogue. Cette boîte principale contient 2 boutons pour naviguer dans le répertoire (partie Contrôleur), elle contient aussi un menu permettant de gérer les fenêtres de vue : le menu `Vues` permet à l'utilisateur d'ouvrir des boîtes de dialogue non modales pour visualiser des informations du répertoire. L'application propose 3 vues différentes : 2 vues pour la fiche courante du répertoire et une pour indiquer la position de cette fiche. La commande `Nouvelle Fiche` ouvre une boîte de dialogue pour saisir et ajouter une nouvelle fiche au répertoire.

Il est clair que cet exemple est un cas d'école qui est très mal conçu du point de vue de l'ergonomie !



8.2 Les modèles de conception (design patterns)

Cet exemple est l'occasion d'utiliser des modèles de conception (design patterns). Il s'agit de modèles objets plus ou moins complexes fournissant des « solutions objet » à des problèmes usuels. On peut dire que les modèles de conception sont à la programmation objet ce que l'algorithmique est à la programmation procédurale, l'objectif étant de ne pas re-concevoir des solutions orientées objet à des problèmes pour lesquels on connaît déja de bonnes solutions, c'est-à-dire faciles à maintenir et faciles à faire évoluer.

Les modèles de conception ne fournissent néanmoins pas la totalité de l'architecture du logiciel, ils proposent une architecture seulement pour certaines sous-parties du logiciel.

Pour l'exemple que l'on souhaite traiter, nous allons utiliser deux modèles de conception :

- le modèle Sujet-Observateurs
- le modèle Singleton

Le modèle Sujet-Observateurs.

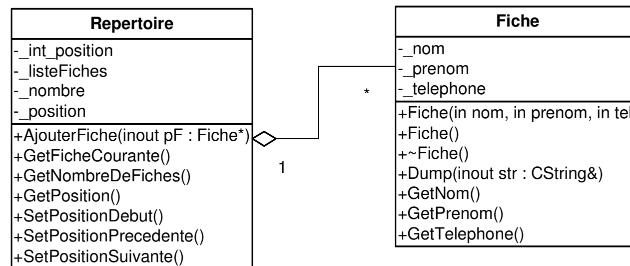
Le principal problème dans les applications multi-vues est que toutes les vues soient informées du changement du modèle. Le modèle de conception Sujet-Observateurs clarifie la relation qu'il y a entre le modèle (ici il s'agit d'une instance de la classe `Repertoire`) et les Vues (ici il s'agit de boîtes de dialogue). L'objet `Repertoire` sera le Sujet, celui-ci aura 0, 1 ou plusieurs Observateurs (les vues).

Le modèle Singleton.

Le modèle Singleton permet à une classe de ne générer qu'une *instance unique*. De plus, cette instance est *accessible depuis tout le programme*. On va donc développer la classe `Repertoire` sur le modèle Singleton pour qu'il y ait un sel objet `Repertoire` dans toute l'application, et que c'est objet soit accessible depuis tout le programme.

8.3 La partie Modèle

Pour notre application, le modèle est constitué d'un objet `Repertoire`. Cet objet `Repertoire` stocke l'adresse d'objets `Fiche` alloués dynamiquement. Le répertoire est responsable de la désallocation de tous les objets `Fiche`. Le répertoire stocke les adresses des objets `Fiche` dans une liste chaînée.




```
--------------------------------------------------------
class Repertoire
{
public:
       void AjouterFiche(Fiche *);
       int GetNombreDeFiches() const;
       Fiche * GetFicheCourante() const;
       void SetPositionDebut();
       void SetPositionSuivante();
       void SetPositionPrecedente();
       int GetPosition() const;
       Repertoire();
       Repertoire(const Repertoire &);
       virtual ~Repertoire();
```

```
private:
	CList<Fiche *,Fiche*> _listeFiches;
	int _nombre;		// nombre de fiches
	int _int_position;	// numéro d'une fiche (-1 si pas de fiche)
	POSITION _position;	// position dans la liste (sorte de pointeur)
};

---------------------------------------------------------------------
// Repertoire.cpp : implémentation

Repertoire::Repertoire()
{
	_int_position=-1;	//-1 indique qu'il n'ya pas de fiche
	_nombre=0;
	_position=NULL;
}

Repertoire::~Repertoire()
{
	// le déstructeur désalloue les objets Fiche
	while(!_listeFiches.IsEmpty())		delete _listeFiches.RemoveHead();
}

void Repertoire::AjouterFiche(Fiche * pF)
{
	_listeFiches.AddTail(pF);		//ajoute une fiche en fin
	_nombre++;
	_position=_listeFiches.GetTailPosition();	//positionne en fin
	_int_position=GetNombreDeFiches()-1;		// numéro de la dernière fiche
}

Fiche * Repertoire::GetFicheCourante() const
{
	if(GetNombreDeFiches()) return _listeFiches.GetAt(_position);
	else return NULL;	// retourne NULL si repertoire vide
}

int Repertoire::GetNombreDeFiches() const{ return _nombre; }

void Repertoire::SetPositionDebut()
{
	_position=_listeFiches.GetHeadPosition();
	if(_position==NULL) _int_position=-1;
	else _int_position=0; // la première fiche est nuémrotée 0
}

void Repertoire::SetPositionSuivante()
{
	if(_position!=NULL){
		_listeFiches.GetNext(_position);
		_int_position++;
	}
	if(_position==NULL){
		_position=_listeFiches.GetTailPosition();
		_int_position=GetNombreDeFiches()-1;
	}
}
```

```
void Repertoire::SetPositionPrecedente()
{
	if(_position!=NULL){
		_listeFiches.GetPrev(_position);
		_int_position--;
	}
	if(_position==NULL){
		_position=_listeFiches.GetHeadPosition();
		if(_position==NULL) _int_position=-1;
		else _int_position=0;
	}
}

int Repertoire::GetPosition() const{return _int_position; }
--------------------------------------------------------------------

--------------------------------------------------------------------
class Fiche
{
public:
	Fiche();
	Fiche(const CString & nom, const CString & prenom, const CString & tel);
	virtual ~Fiche();

	CString GetNom() const{ return _nom;}
	CString GetPrenom() const{ return _prenom;}
	CString GetTelephone() const{ return _telephone;}

	// affiche les attributs dans une chaîne
	virtual void Dump(CString & str) const;
private:
	CString _nom;
	CString _prenom;
	CString _telephone;
};
--------------------------------------------------------------------
```

8.4 Modifier la classe `Repertoire` suivant le modèle de conception Singleton

Singleton
#_pInstance : *Singleton
#Singleton(in : Singletonconst &) #Singleton() #~Singleton() +Instance() +LibererInstance()

Développer une classe suivant le modèle de conception Singleton revient à

- mettre les constructeurs et destructeur dans la partie protégée de la classe
- mettre un pointeur statique dans la classe (pointeur non public)
- mettre deux méthodes statiques pour obtenir l'instance unique et désallouer cette instance

Une donnée membre statique n'existe qu'en un seul exemplaire pour la classe (contrairement aux données non statiques qui sont mémorisées dans chaque objet). Le pointeur `_pInstance` va stocker l'adresse de l'unique objet de la classe `Singleton`. Les constructeurs et destructeurs sont déclarés protégés pour que l'utilisateur ne puisse pas créer d'objet automatique (dans la pile), d'ailleurs l'utilisateur ne peut pas non plus allouer d'objet dans le tas avec la syntaxe **`ptr=new Singleton`**. L'allocation dynamique ne sera possible que depuis une méthode de la classe!

L'utilisateur doit donc utiliser la méthode statique **`Singleton::Instance()`** pour obtenir l'adresse de l'objet unique (une méthode statique peut être invoquée sans mentionner d'objet courant). Au premier appel de cette méthode, celle-ci alloue une instance dans le tas puis retourne son adresse. Pour les appels suivants, la méthode retourne l'adresse de l'instance existante.

L'utilisateur doit appeler la méthode statique **`Singleton::LibererInstance()`** pour libérer l'instance lorsque le singleton n'a plus à être utilisé (typiquement en fin de programme).

```
------------------------------------------------
//Singleton.h
class Singleton
{
public:
	static Singleton * Instance();
	static void LibererInstance();
protected:
	Singleton();
	virtual ~Singleton();

	/* il n'est pas nécessaire de définir le constructeur de copie. En revanche
	il faut le déclarer pour annuler le contructeur de copie par défaut */
	Singleton(const Singleton &);

	static Singleton * _pInstance;	// pour stocker l'adresse de l'instance unique
};
------------------------------------------------
------------------------------------------------
//Singleton.cpp
	...
#include "Singleton.h"

Singleton * Singleton::_pInstance=NULL; // initialisation du pointeur statique

Singleton::Singleton(){ // ... code d'initialisation ... }

Singleton::~Singleton(){  }

Singleton * Singleton::Instance(){  // retourne l'instance unique
	if(_pInstance==NULL) _pInstance=new Singleton;
	return _pInstance;
}

void Singleton::LibererInstance(){  //désalloue l'instance unique
	delete _pInstance;
	pInstance=NULL;
}
------------------------------------------------
```

Modifier la classe **`Repertoire`** suivant ce modèle pour qu'un seul objet de la classe `Repertoire` soit généré.

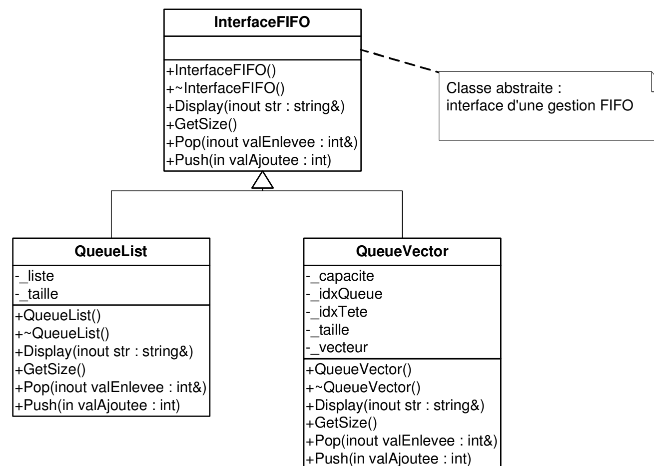
Repertoire
-_instance -_int_position -_listeFiches -_nombre -_position
#Repertoire(in p0 : Repertoireconst &) #Repertoire() #~Repertoire() +AjouterFiche(inout pF : Fiche*) +DeleteInstance() +GetFicheCourante() +GetInstance() +GetNombreDeFiches() +GetPosition() +SetPositionDebut() +SetPositionPrecedente() +SetPositionSuivante()

8.5 Rappel sur les classes abstraites (notion d'interface)

En C++, la notion de *classe d'interface* se réalise sous la forme de classes dites *abstraites*. Une classe C++ est abstraite si au moins une de ses méthodes est déclarée *virtuelle pure* (son prototype se termine par =0), c'est-à-dire une méthode pour laquelle la ligature dynamique est réalisée (une table de méthodes virtuelles est dans ce cas mise en oeuvre pour que la méthode réelle soit retrouvée dynamiquement à l'exécution) mais pour laquelle aucune implémentation n'est donnée dans la classe de base. Une telle classe ne peut donc pas engendrer d'objet. L'implémentation des méthodes virtuelles pures est réalisée dans la (ou les) classe(s) dérivée(s).

En conséquence, on dit qu'une classe dérivée d'une classe abstraite *implémente l'interface* définie par la classe abstraite si elle implémente toutes les méthodes virtuelles pures de la classe de base abstraite. Si une des méthodes virtuelles pures n'est pas implémentée dans la classe dérivée, la classe dérivée reste abstraite (et ne peut pas créer d'objet). La classe qui implémente l'interface définie par une classe abstraite est dit classe *concrète*.

A titre d'exemple, on peut définir l'interface d'un conteneur d'entiers de type **FIFO** (ce qu'on appelle une "file" en français, et "queue" en anglais). Un tel conteneur a un comportement assez simple : l'ajout et le retrait. En plus, pour les besoins de l'affichage, on va permettre au conteneur d'afficher son contenu dans une chaîne de caractères de type **`CString`**. Le diagramme de classes est le suivant :



Le comportement d'un conteneur FIFO se résume donc à l'interface (la classe abstraite) ci-dessous :

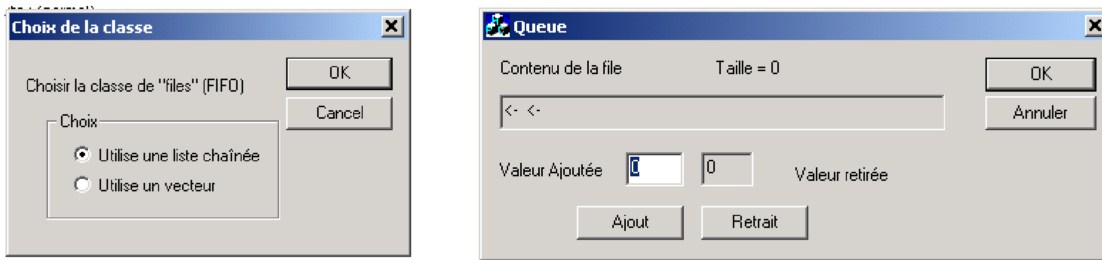
```
class InterfaceFIFO        // classe abstraite
{
public:
      InterfaceFIFO(){}                //déclaration+définition du constructeur
      virtual ~InterfaceFIFO(){}       //déclaration+définition du destructeur

      // méthodes virtuelles pures (pas d'implémentation)
      virtual void Push(int valeurEjoutee)=0;
      virtual bool Pop(int & valeurEnlevee)=0;
      virtual unsigned GetSize() const=0;
      virtual void Display(CString & str) const =0;
};
```

Note : ici, toute la classe abstraite peut être définie au moyen d'un seul fichier `InterfaceFIFO.h`.

Ensuite, il y a différentes façons d'implémenter cette interface. Soit en utilisant une liste chaînée - un objet `list<int>` (STL) ou un objet `CList<int,int>` (MFC) - ou en utilisant un tableau - `vector<int>` ou `CArray<int,int>`-. La première implémentation est la classe `QueueList`, la seconde la classe `QueueVector`.

Important : peu importe l'implémentation réalisée, l'utilisation d'un objet dérivé de l'interface **`InterfaceFIFO`** peut toujours se faire via un pointeur de type **`InterfaceFIFO*`**. Le nom de la classe concrète ne va être utilisé que pour instancier les objets. Il sera donc facile de remplacer une classe concrète par une autre
Le programme partiel ci-dessous présente l'utilisation de l'interface `InterfaceFIFO` et des classes concrètes implémentant cette interface. L'utilisation de cette interface s'effectue au sein d'une application MFC de type

"Boîte de Dialogue". Tout d'abord, l'utilisateur choisit l'implémentation de l'interface, puis le programme d'utilisation commence.

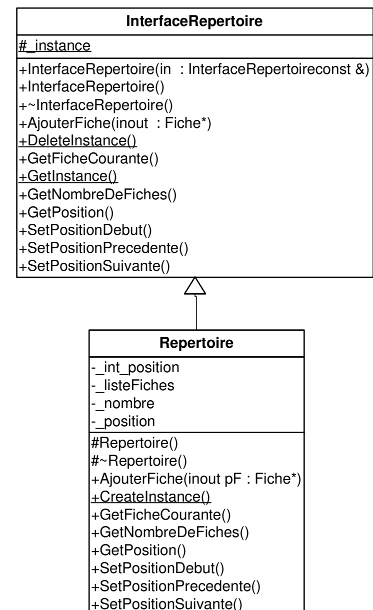


La boîte de dialogue principale contient le code suivant (vue partielle). Tout le dialogue utilisateur n'utilise que les méthodes de l'interface **InterfaceFIFO**.

```
--------------------------------------------------------------------------------
//QueueDlg.cpp : implémentation de la boîte de dialogue principale
CQueueDlg::CQueueDlg(CWnd* pParent):CDialog(CQueueDlg::IDD, pParent){
	// ...
	pQueue=NULL;		// CQueueDlg::pQueue est un membre de type InterfaceFIFO *
}

CQueueDlg::~CQueueDlg(){
	delete pQueue;		// désalloue l'objet FIFO
}

// cette méthode s'exécute à l'ouverture de la boîte de dialogue
BOOL CQueueDlg::OnInitDialog(){
	...
	// boîte de dialogue pour le choix utilisateur (capture écran ci-dessus)
	CChoixDlg dlg;
	int choix=0;
	if(dlg.DoModal()==IDOK)	choix=dlg.m_choix_classe;

	// selon le choix, crée un objet dynamique QueueVector ou QueueList.
	if (choix==0) pQueue=new QueueList;
	else pQueue=new QueueVector;
	MAJ_Affichage();	// met à jour l'affichage
	return TRUE;
}
void CQueueDlg::OnClickAjout(){
	UpdateData(TRUE);
	if(pQueue!=NULL){
		pQueue->Push(m_ajout);
		MAJ_Affichage();
	}
}
void CQueueDlg::OnClickRetrait(){
	if(pQueue!=NULL){
		pQueue->Pop(m_retrait);
		MAJ_Affichage();
	}
}
void CQueueDlg::MAJ_Affichage(){
	if(pQueue!=NULL){
		CString str;
		str.Format("Taille = %d",pQueue->GetSize());
		m_str_taille=str;
		pQueue->Display(m_edit_queue);
		UpdateData(FALSE);
	}
}
--------------------------------------------------------------------------------
```

8.6 Interface de classe `Repertoire`

Au vu de ce qui précède, il sera plus facile de faire évoluer l'application si une classe abstraite d'interface est définie pour le répertoire. Pour pouvoir bénéficier de la notion d'interface et du modèle de conception Singleton, une solution est la suivante.



On peut définir une classe abstraite `InterfaceRepertoire` contenant toutes les méthodes que le répertoire devra concrétement implémenter. En outre, on peut ajouter (pour le modèle Singleton)

- un pointeur statique de type **`InterfaceRepertoire *`**
- une méthode statique **`GetInstance()`** retournant la valeur du pointeur
- une méthode statique **`DeleteInstance()`** pour libérer l'instance unique

Dans ce cas, la création de l'instance unique ne peut pas être effectuée dans la classe de base abstraite (puisque la classe abstraite ne peut pas connaître le type de la (des) classe(s) concrète(s) qui en dérive(nt) !)

Donc, la création de l'instance unique est reléguée à la classe d'implémentation de l'interface (ici dans la classe `Repertoire`). La classe concrète `Repertoire` contient une méthode statique **`CreateInstance()`** pour allouer un objet unique de type `Repertoire`. Le pointeur `_instance` de type `InterfaceRepertoire *` doit donc être visible depuis la classe dérivée, c'est pourquoi il doit être déclaré `protected` dans la classe abstraite.

En C++, la classe abstraite pour la couche métier et l'implémenattion de ce répertoire sont donnés ci-dessous.

```
----------------------------------------------------
// InterfaceReprtoire.h : classe abstraite pour la couche métier

class InterfaceRepertoire
{
public:
	InterfaceRepertoire(){}
	InterfaceRepertoire(const InterfaceRepertoire &){}
	virtual ~InterfaceRepertoire(){}

	//méthodes virtuelles pures
	virtual void AjouterFiche(Fiche *)=0;
	virtual int GetNombreDeFiches() const=0;
	virtual Fiche * GetFicheCourante() const=0;
	virtual void SetPositionDebut()=0;
	virtual void SetPositionSuivante()=0;
	virtual void SetPositionPrecedente()=0;
	virtual int GetPosition() const=0;

	// modèle singleton
	static InterfaceRepertoire * GetInstance();
	static void DeleteInstance();
protected:
	static InterfaceRepertoire * _instance;
};

----------------------------------------------------
```

```
-----------------------------------------------------
//InterfaceRepertoire.cpp : implémentation du modèle singleton
      ...
#include "InterfaceRepertoire.h"

InterfaceRepertoire * InterfaceRepertoire::_instance=NULL;

InterfaceRepertoire* InterfaceRepertoire::GetInstance()
{
      return _instance;
}

void InterfaceRepertoire::DeleteInstance()
{
      if(_instance!=NULL)
      {
             delete _instance;
             _instance=NULL;
      }
}
-----------------------------------------------------
```

```
-----------------------------------------------------
//Repertoire.h : classe Repertoire implémentant InterfaceRepertoire

#include "InterfaceRepertoire.h"
#include<afxtempl.h> // pour l'utilisation de la classe template CList<>

class Repertoire : public InterfaceRepertoire
{
protected:
      // empecher les instances multiples (suivant le modèle Singleton)
      Repertoire();
      Repertoire(const Repertoire &);
      virtual ~Repertoire();
public:
      // méthode statique pour générer l'instance unique
      static void CreateInstance();

      // implémentation de la classe abstraite d'interface
      void AjouterFiche(Fiche * pF);
      int GetNombreDeFiches() const;
      Fiche * GetFicheCourante() const;
      void SetPositionDebut();
      void SetPositionSuivante();
      void SetPositionPrecedente();
      int GetPosition() const;
private:
      CList<Fiche *,Fiche*> _listeFiches;
      int _nombre;
      int _int_position;
      POSITION _position;
};
-----------------------------------------------------
```

```
----------------------------------------------------
// Repertoire.cpp
        ...

#include "Repertoire.h"

void Repertoire::CreateInstance()
{
        if(_instance==NULL) _instance=new Repertoire;
}


Repertoire::Repertoire()
{
        _int_position=-1;
        _nombre=0;
        _position=NULL;
}

        ...             // l'implémentation donnée précédemment


----------------------------------------------------
```

Pour cette solution utilisant une classe abstraite, la création de l'instance unique de la classe concrète doit être effectuée en appelant la méthode **`Repertoire::CreateInstance()`**. C'est normalement la seule fois où le nom de la classe implémentant l'interface doive être utilisé. Pour tous les autres accès à l'objet répertoire, on peut utiliser la méthode statique **`InterfaceRepertoire::GetInstance()`** de la classe abstraite.

Dans une application MFC, on peut par exemple créer l'instance au niveau du constructeur de l'objet application (ou dans la méthode **`C___App::InitInstance()`**) et libérer l'instance au niveau du destructeur de l'objet application (ou dans la méthode **`C___App::ExitInstance()`**).

```
CMVCRepertoireApp::CMVCRepertoireApp()
{
        // la seule fois où le nom de la classe concrète Repertoire est utilisé
        Repertoire::CreateInstance();
}

CMVCRepertoireApp::~CMVCRepertoireApp()
{
        InterfaceRepertoire::DeleteInstance();
}
```

8.7 Modifier la classe `Fiche` pour que seuls des objets alloués dans le tas puissent être créés

Compte tenu du choix de conception, le `Repertoire` stocke des adresses d'objets et se charge (au niveau du destructeur) de leur désallocation. Il est clair que le `Repertoire` ne doit donc stocker que des adresses d'objets alloués dans le tas et surtout pas d'adresses d'objets automatique.

Il est donc souhaitable de modifier la classe `Fiche` pour empêcher la création d'objets automatiques. En s'inspirant du modèle de conception Singleton, on voit qu'il est possible de ne manipuler que des instances allouées dans le tas si :

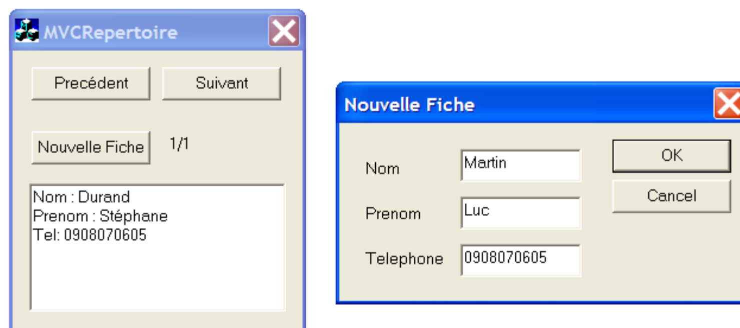
- les constructeurs de la classe `Fiche` sont déclarés `protected`
- une méthode statique alloue un nouvel objet de la classe `Fiche` dans le tas et retourne son adresse.

Dans ce cas, les objets de la classe `Fiche` peuvent toujours être désalloués par l'utilisateur en utilisant l'opérateur `delete`.

```
---------------------------------------------------------------------------
// Fiche.h : déclaration de la classe Fiche pour empêcher les objets automatiques
class Fiche
{
protected:
	Fiche();
	Fiche(const Fiche &);
	Fiche(const CString & nom, const CString & prenom, const CString & tel);
public:
	// méthode statique qui retourne un objet alloué dans le tas
	static Fiche * AllouerObjet(const CString & nom,const CString & prenom,
	                                                  const CString & tel);
	virtual ~Fiche();
	CString GetNom() const{ return _nom;}
	CString GetPrenom() const{ return _prenom;}
	CString GetTelephone() const{ return _telephone;}
	virtual void Dump(CString & str) const;
private:
	CString _nom;
	CString _prenom;
	CString _telephone;
};
---------------------------------------------------------------------------
---------------------------------------------------------------------------
//Fiche.cpp
	...

Fiche * Fiche::AllouerObjet(const CString & nom, const CString & prenom,
                                                 const CString & tel){
	return new Fiche(nom,prenom,tel);
}
---------------------------------------------------------------------------
```

8.8 Réaliser une application de type boîte de dialogue utilisant le singleton `Repertoire`

En utilisant les classes métier définies précédemment sur le modèle de conception Singleton, écrire une application qui affiche dans la boîte principale la fiche courante, la position et le nombre de fiches du répertoire. Le bouton « `Nouvelle Fiche` » ouvre une boîte modale de saisie d'une nouvelle `Fiche`.

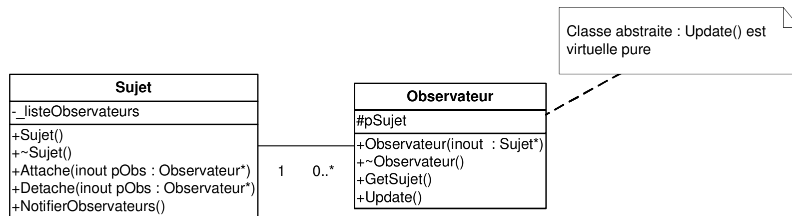


8.9 Le modèle de conception Sujet-Observateurs

Nous allons faire évoluer l'application précédente pour que les données affichées à l'utilisateur puissent être réparties sur plusieurs fenêtres différentes. L'application va donc contenir un modèle mais plusieurs vues.

Pour celà, il convient d'isoler la relation particulière entre le *sujet* (le modèle) et ses *observateurs* (les boîtes de dialogue de vue) dans des classes de base. Le diagramme de classe suivant illustre le lien entre le sujet et ses observateurs.

Un sujet stocke les adresses de tous ses observateurs. Ainsi, le sujet est capable d'informer (Notifier) ses observateurs que son état a changé. Lorsque l'état du sujet change, celui-ci invoque la méthode `Observateur::Update()` (c'est une méthode polymorphe) de tous ses observateurs.
Les observateurs sont donc informés des changements d'état du sujet. L'implémentation de la méthode `Update()` des observateurs consiste alors à demander au sujet son nouvel état.



Les classes de base du modèle de conception Sujet-Observateurs sont les suivantes.

Note : chaque classe possède un pointeur sur un objet de l'autre classe. Techniquement, il faut donc faire précéder la définition des classes d'une *déclaration anticipée* de l'autre classe.

```
--------------------------------------------------
// Observateur.h : classe abstraite d'interface d'un observateur
#include "sujet.h"

class Sujet;         //déclaration anticipée

class Observateur
{
public:
      Observateur(Sujet * pS=NULL){ pSujet=pS;}
      virtual ~Observateur(){}

      virtual void Update()=0;
      Sujet * GetSujet() const{     return pSujet;}
protected:
      Sujet * pSujet;       //un observateur stocke l'adresse de son sujet
};
--------------------------------------------------

--------------------------------------------------
//Sujet.h : classe de base d'un Sujet
#include<afxtempl.h>
#include "Observateur.h"

class Observateur;          //déclaration anticipée

class Sujet
{
public:
      Sujet();
      virtual ~Sujet();
      void NotifierObservateurs() const;
      bool Attache(Observateur * pObs);
      bool Detache(Observateur * pObs);
private:
      CList<Observateur *,Observateur *> _listeObservateurs; //liste d'observateurs
};
--------------------------------------------------
```

```
-------------------------------------------------
//sujet.cpp : implementation de la classe de base Sujet

Sujet::Sujet(){}

Sujet::~Sujet(){}

// méthode invoquée pour informer tous les observateurs d'un changement
// d'état du sujet
void Sujet::NotifierObservateurs() const
{
	// les adresses des observateurs sont stockées dans _listeObservateurs
	POSITION p=_listeObservateurs.GetHeadPosition();
	while(p) _listeObservateurs.GetNext(p)->Update();
}

// ajouter un nouvel observateur (sauf s'il est déja dans la liste)
bool Sujet::Attache(Observateur * pObs)
{
	bool ajout=true;
	POSITION p=_listeObservateurs.GetHeadPosition();
	while(p!=NULL && ajout )
	{
		if(_listeObservateurs.GetNext(p)==pObs) ajout=false;
	}
	if(ajout) _listeObservateurs.AddTail(pObs);
	return ajout;
}

// supprimer un observateur de la liste
bool Sujet::Detache(Observateur * pObs)
{
	bool retrait=false;
	POSITION p=_listeObservateurs.GetHeadPosition();
	while(p!=NULL && !retrait )
	{
		if(_listeObservateurs.GetAt(p)==pObs)
		{
			retrait=true;
			_listeObservateurs.RemoveAt(p);
		}
		else _listeObservateurs.GetNext(p);
	}
	return retrait;
}
-------------------------------------------------
```

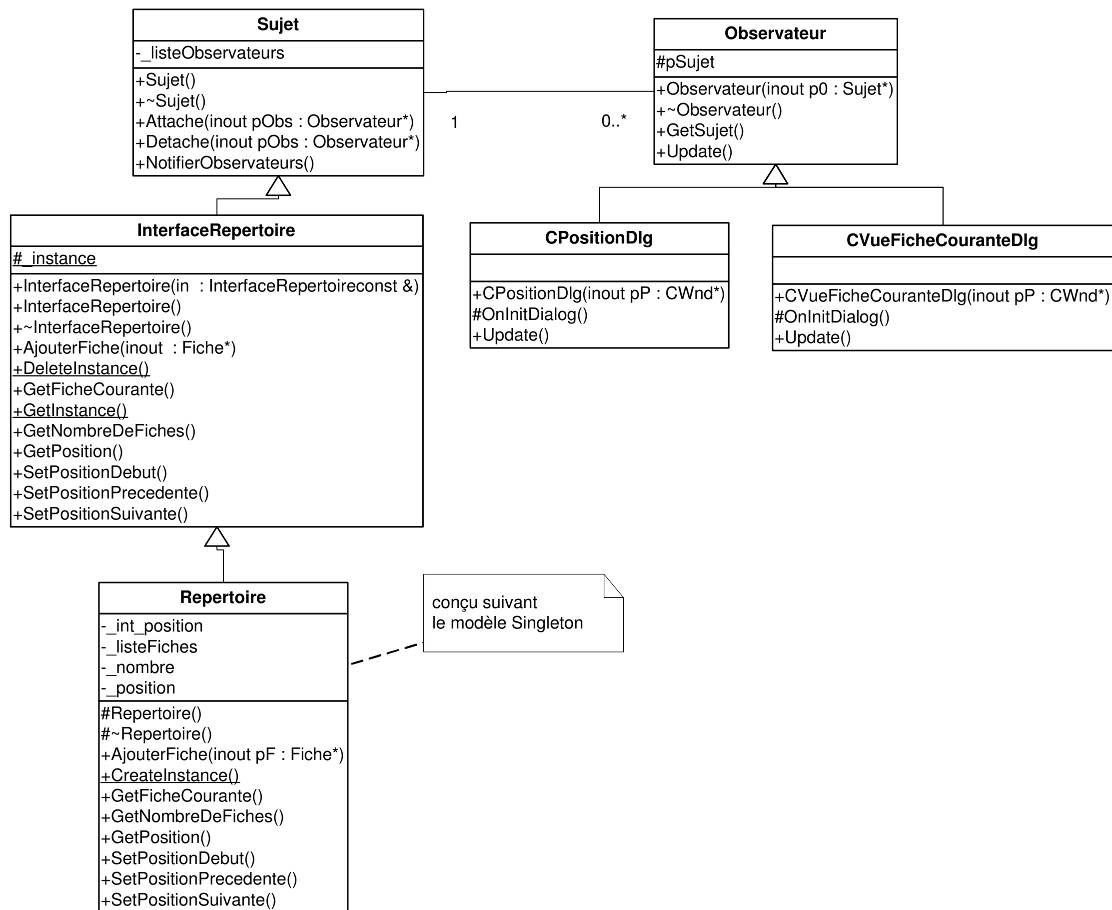
L'utilisation du modèle de conception Sujet-Observateurs se fait par dérivation.

Dans notre exemple, la classe `InterfaceRepertoire` (classe abstraite intégrant le modèle Singleton) doit être dérivée de la classe `Sujet`, ceci pour bénéficier de la gestion des observateurs (attacher/détacher un observateur, notifier les observateurs).

Ensuite, les classes de vue doivent dériver de la classe abstraite `Observateur`. Chaque classe de vue doit implémenter le traitement adapté pour la méthode `Update()`.

Dans notre exemple, ce sont des boîtes de dialogue non modales qui jouent le rôle de « vues ». Toutes les boîtes de dialogue de vue doivent donc être dérivées de la classe abstraite `Observateur`, et doivent donc implémenter la méthode `Update()`, faute de quoi ces classes de vues restent abstraites. Pour ces classes, l'héritage multiple est

utilisé puisque les classes de boîtes de dialogue dérivent déjà de la classe `CDialog`. L'application conçue sur le modèle de conception Sujet-Observateurs suit donc l'architecture de classes suivante :



8.10 Application finale

Pour obtenir l'application finale avec plusieurs vues, on peut reprendre l'application précédente. Les modifications sont les suivantes :

- développer/ajouter les classes de base `Sujet`/`Observateur`
- mettre la classe `Sujet` comme classe de base de la classe `InterfaceRepertoire`

```
#include "Fiche.h"
#include<afxtempl.h>
#include "Sujet.h"
class InterfaceRepertoire : public Sujet
{
public:
	virtual void AjouterFiche(Fiche *)=0;
	virtual int GetNombreDeFiches() const=0;
	…
};
```

- créer des ressources et des classes de boîte de dialogue pour les différentes vues. Par exemple, **`CPositionDlg, CFicheCouranteVue1Dlg, CFicheCouranteVue2Dlg.`**

- préciser que ces classes de boîtes de dialogue sont des implémentations de la classe abstraite **`Observateur`**

```
-------------------------------------------------------
#include "Observateur.h"

//la classe CPositionDlg dérive à la fois de CDialog et de Observateur
class CPositionDlg : public CDialog, public Observateur
{
public:
	CPositionDlg(CWnd* pParent = NULL);   // standard constructor
	void Update(); // à definir car virtuelle pure dans la classe Observateur
	…
};
-------------------------------------------------------
```

- il faut compléter l'implémentation de la boîte de dialogue pour prendre en compte l'architecture Sujet-Observateurs

```
--------------------------------------------------------------------------
// implémentation de CPositionDlg
#include "Repertoire.h"
#include "PositionDlg.h"

// Le sujet de cet observateur est l'instance unique de la classe Repertoire
CPositionDlg::CPositionDlg(CWnd* pParent):CDialog(CPositionDlg::IDD, pParent),
                                    Observateur(InterfaceRepertoire::GetInstance())
{
	//{{AFX_DATA_INIT(CPositionDlg)
	m_position_string = _T("");   // chaîne affichant la position
	//}}AFX_DATA_INIT

	/* attacher cet observateur au Sujet, sans quoi cette boîte ne
	 recevra pas les messages de notification Update() !  */

	GetSujet()->Attache(this);
}

void CPositionDlg::Update()  //implémentation de la méthode Update()
{
	// Le sujet (de type Sujet*) est converti en InterfaceRepertoire*
	InterfaceRepertoire * pR=static_cast<InterfaceRepertoire*>(GetSujet());

	m_position_string.Format("Position = %d/%d",pR->GetPosition()+1,
	                                              pR->GetNombreDeFiches());
	UpdateData(FALSE);
}

// la boîte de dialogue demande l'état du sujet dès l'ouverture
BOOL CPositionDlg::OnInitDialog()
{
	CDialog::OnInitDialog();
	Update();      //Met à jour la boîte de dialogue à partir du sujet
	return TRUE;
}

	...

--------------------------------------------------------------------------
```

La bibliothèque MFC a prévu un mécanisme pour assurer la persistance d'objets (sauvegarde et resitution d'objets dans des archives). Ce mécanisme est appelé *sérialisation*. La sérialisation repose sur la possibilité qu'ont les objets de s'enregistrer dans une archive ou de se reconstituer depuis une archive. L'archive est un objet qui assure la liaison avec le support de sauvegarde physique, c'est-à-dire un fichier.

Le mécanisme de sérialisation n'est disponible que pour les objets dérivés de la classe de base `CObject`. En effet, la classe `CObject` propose des mécanismes nécessaires à la mise en oeuvre de la sérialisation, notamment la reconnaissance de type à l'exécution (Run-Time class information).

Remarque : il est à noter que les MFC n'utilisent pas le mécanisme RTTI du C++ pour la reconnaissance de type. La biobliothèque MFC implémente la reconnaissance de type via des macros (revoir la section 1 du cours).

L'utilisation du mécanisme de sérialisation impose donc que les objets que l'on souhaite sauvegarder soient issus de classes dérivées de **`CObject`**. En outre, deux macros doivent être appelées pour compléter le mécanisme de sérialisation : **`DECLARE_SERIAL(ClassName)`** dans la déclaration de la classe (fichier header) et **`IMPLEMENT_SERIAL(ClassName,BaseClass,numVersion)`** dans le fichier de définition de la classe (fichier .cpp).

`CObject::Serialize()` : la lecture/l'écriture d'un objet (dérivé de `CObject`) dans une archive sont programmées dans la surdéfinition de la méthode **`Serialize()`** spécifique à la classe (noter que cette méthode est virtuelle dans la classe `CObject`).

Si l'on reprend l'application de stockage de numéros de téléphone de la section précédente, il faut ajouter le mécanisme de sérialisation à la classe `Fiche` et à la classe `Repertoire`.

```
------------------------------------------------------------------------
// Fiche.h : classe d'objets persistants

#include<afx.h>
class Fiche : public CObject
{
	DECLARE_SERIAL(Fiche) //ajout de methodes permettant la sérialisation
public:
	virtual void Serialize(CArchive & ar);

protected:
	Fiche();
	Fiche(const CString & nom, const CString & prenom, const CString & tel);
	Fiche(const Fiche &);
public:

	static Fiche * AllouerObjet(const CString & nom, const CString & prenom,
					const CString & tel);
	virtual ~Fiche();
	CString GetNom() const{ return _nom;}
	CString GetPrenom() const{ return _prenom;}
	CString GetTelephone() const{ return _telephone;}
	virtual void Dump(CString & str) const;
private:
	CString _nom;
	CString _prenom;
	CString _telephone;
};
------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------
// Fiche.cpp: implementation of the Fiche class.
#include "stdafx.h"
#include "MVCRepertoire.h"
#include "Fiche.h"

      IMPLEMENT_SERIAL(Fiche,CObject,0)    //version 0

// on indique dans cette méthode les données qui doivent persister
void Fiche::Serialize(CArchive & ar)
{
      CObject::Serialize(ar);        // appeler la méthode héritée

      if(ar.IsStoring())             // si écriture dans l'archive
      {
            ar <<_nom;               // sauvegarde des attributs (CString) dans l'archive
            ar <<_prenom;
            ar <<_telephone;
      }
      else                           // si lecture depuis l'archive
      {
            ar >>_nom;               // lecture des objets CString depuis l'archive
            ar >>_prenom;
            ar >>_telephone;
      }
}

Fiche::Fiche(){}

Fiche::Fiche(const CString & nom,
             const CString & prenom,
             const CString & tel):_nom(nom),_prenom(prenom),_telephone(tel){}

Fiche::~Fiche(){}

Fiche * Fiche::AllouerObjet(const CString & nom, const CString & prenom,
                                                    const CString & tel){
      return new Fiche(nom,prenom,tel);
}
void Fiche::Dump(CString & str)const{
      str="Nom : " +_nom + "\r\nPrenom : " + _prenom + "\r\nTel: " + _telephone;
}
---------------------------------------------------------------------------
```

Noter : les types primitifs et certains types courants peuvent être écrits ou lus dans une archive avec les opérateurs << et >>. C'est le cas notamment des objets `CString`. (voir ci-dessus)

Il faut également compléter la hiérarchie de classes `Repertoire` pour ajouter la persistance à l'objet répertoire. Tout d'abord, ajouter **`CObject`** comme classe de base de la classe abstraite **`InterfaceRepertoire`**.

```
---------------------------------------------------------------------------
#include "Fiche.h"
#include "Sujet.h"
#include<afx.h>       //utilisation de la classe CObject

class InterfaceRepertoire : public Sujet, public CObject
{     ...
      // pas d'autre modification pour la classe abstraite
};
---------------------------------------------------------------------------
```

Ensuite, il faut compléter la classe concrète **Repertoire** par l'ajout de la surdéfinition de la méthode **Serialize()** et par l'ajout de l'appel des macros **DECLARE_SERIAL()** et **IMPLEMENT_SERIAL()**.

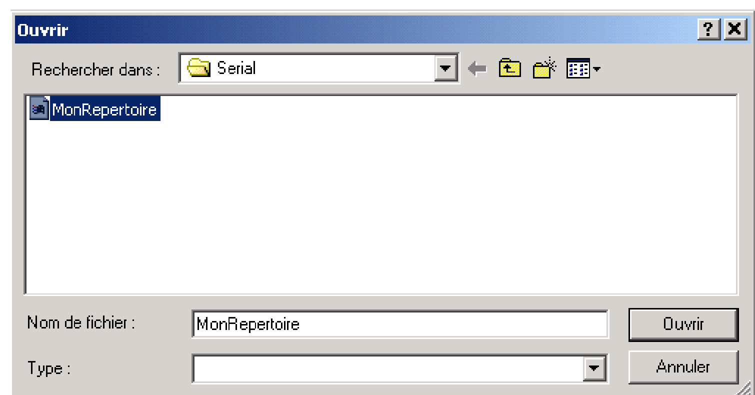
Il faut en outre remplacer l'objet membre de type **CList<Fiche*,Fiche*>** par un objet membre de type **CObList** (liste de pointeurs sur des objets dérivés de CObject) ou par un objet de type **CTypedPtrList<CObList,Fiche*>**. Ces deux derniers types implémentent la gestion d'une liste chaînée, avec les mêmes méthodes que CList, mais avec en plus une méthode **Serialize()** permettant à tout le conteneur de se lire/ou s'écrire dans une archive. Bien évidemment, les objets du conteneur doivent être des objets dérivés de **CObject**.

```
--------------------------------------------------------------------------
#include "InterfaceRepertoire.h"
#include<afxtempl.h>

class Repertoire : public InterfaceRepertoire
{
	DECLARE_SERIAL(Repertoire)
public:
	virtual void Serialize(CArchive & ar);
protected:
	Repertoire();
	virtual ~Repertoire();
public:
	void AjouterFiche(Fiche * pF);
	int GetNombreDeFiches() const;
	Fiche * GetFicheCourante() const;
	void SetPositionDebut();
	void SetPositionSuivante();
	void SetPositionPrecedente();
	int GetPosition() const;
	void Vider();

	static void CreateInstance(); // crée l'instance unique
private:
	CObList _listeFiches;                         // modification
	// CTypedPtrList<CObList,Fiche*> _listeFiches; // possible aussi
	int _nombre;
	int _int_position;
	POSITION _position;
};
--------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------
// RepertoireC.cpp: implementation of the RepertoireC class.
#include "stdafx.h"
#include "MVCRepertoire.h"
#include "Repertoire.h"

	...

	// ici, il ne faut pas mettre InterfaceRepertoire comme classe de Base
	// car il s'agit d'une classe abstraite qui ne peut pas créer d'objet !!!
	IMPLEMENT_SERIAL(Repertoire,CObject,0)

void Repertoire::CreateInstance()
{
	if(_instance==NULL) _instance=new Repertoire;
}
```

```
// sérialisation du répertoire
void Repertoire::Serialize(CArchive & ar)
{
	CObject::Serialize(ar);

	if(ar.IsStoring())	// si écriture dans l'archive
	{
		_listeFiches.Serialize(ar);
		ar<<_nombre;
		// la position ne peut pas être enregistrée
	}
	else{		// si lecture de l'archive
		Vider();		//vider le répertoire
		_listeFiches.Serialize(ar); // lire la liste depuis l'archive
		ar>>_nombre;

			// mettre la position en fin
		_int_position=GetNombreDeFiches()-1;
		_position=_listeFiches.GetTailPosition();

			// indiquer aux observateurs que le Sujet a changé
		NotifierObservateurs();
	}
}

void Repertoire::Vider(){
	while(!_listeFiches.IsEmpty())	delete _listeFiches.RemoveHead();
	_int_position=-1;
	_position=NULL;
}

Repertoire::Repertoire(){
	_int_position=-1;
	_nombre=0;
	_position=NULL;
}

Repertoire::~Repertoire(){
	Vider();
}

	... // le reste est identique
---------------------------------------------------------------------------
```

Enfin, on peut ajouter un sous-menu `Fichier` à l'application pour pouvoir **`Ouvrir`** ou **`Enregistrer`** le répertoire (via la sérialisation du répertoire).



L'ouverture et l'enregistrement de fichiers disposent de boîtes de dialogue « clé en main » (**`CFileDialog`**). Ces boîtes de dialogue assurent l'affichage, la navigation dans les fichiers etc. Le code des gestionnaires d'événements des commandes **`Ouvrir`** et **`Enregistrer sous`** est donné ci-dessous

```
//gestionnaire de la commande Ouvrir du sous-menu Fichier
void CMVCRepertoireDlg::OnFichierOuvrir(){
	CFileDialog dlg(true);// true -> boîte pour l'ouverture d'un fichier

	if(dlg.DoModal()==IDOK){
		CFileException e;	// objet exception
		CFile mFile;		// fichier
		// ouverture en lecture
		mFile.Open(dlg.GetFileName(),CFile::modeRead,&e);
		// création de l'archive (en mode load) associée au fichier
		CArchive ar(&mFile,CArchive::load);
		InterfaceRepertoire::GetInstance()->Serialize(ar);
	}
}
//gestionnaire de la commande Enregistrer sous du sous-menu Fichier
void CMVCRepertoireDlg::OnFichierEnbregistrersous(){
	CFileDialog dlg(false); // false -> boîte pour l'enregistrement d'un fichier

	if(dlg.DoModal()==IDOK){
		CFileException e;
		CFile mFile;
		mFile.Open(dlg.GetFileName(),CFile::modeCreate|CFile::modeWrite,&e);
		CArchive ar(&mFile,CArchive::store);
		InterfaceRepertoire::GetInstance()->Serialize(ar);
	}
}
```

10 Les applications MFC utilisant le modèle Document-Vue

10.1 Introduction

Jusqu'ici, les applications ont été programmées suivant le modèle `"Dialog Based"` du générateur d'application `AppWizard`. Dans ce modèle, l'application MFC repose initialement sur deux classes : la classe d'application **`CNomdeprojetApp`** dérivée de **`CwinApp`** et la classe de boîte de dialogue **`CNomdeprojetDlg`** dérivée de **`CDialog.`**

Dans ce modèle, la méthode `BOOL CNomdeprojetApp::InitInstance()` crée une boîte de dialogue modale et toute l'interface (Interface Homme Machine ou IHM) avec l'utilisateur est programmée dans la classe de cette boîte de dialogue.

```
BOOL CNomdeprojetApp::InitInstance()
{
        CNomdeprojetDlg dlg;     // objet gérant la boîte de dialogue principale
        m_pMainWnd = &dlg;
        int nResponse = dlg.DoModal();    //création de la fenêtre de type "boîte modale".
        if (nResponse == IDOK){   /* … */  }
        else if (nResponse == IDCANCEL){ /* …  */ }
        return FALSE;
}
```

Même si ce modèle n'utilise que peu de classes pour la partie IHM, il met néanmoins en évidence toutes les techniques de programmation liées à la bibliothèque MFC et aux assistants : mise en place de gestionnaires d'événements, création de boîtes de dialogue pour la saise de données, exploitation des contrôles, exploitation de menus ...

Les applications MFC reposant sur l'architecture Document-Vue vont utiliser les mêmes techniques de programmation, mais la partie IHM va nécessiter plus de classes. Le générateur d'application `AppWizard` va donc générer un canevas d'application comportant initialement plus de classes. Il sera donc nécessaire de comprendre le rôle de chaque classe au sein de l'application.

10.2 Procédure de génération d'une application SDI avec AppWizard

Nous allons générer une application ***Single Document Interface*** (SDI) ayant comme vue une ***forme***, c'est-à-dire une vue permettant d'héberger (comme les boîtes de dialogue) des fenêtres de contrôle (boutons, zones d'édition ...) et des contrôles ActiveX.

1 - Lancer l'assistant MFC AppWizard (.exe) **Nom de projet : SDI**

2 - MFC AppWizard Step 1 : sélectionner le modèle **`Single Document`** **+** l'architecture **`Document/View`**

3 - MFC AppWizard Step 2/6 : laisser les options par défaut (pas d'exploitation de bases de données)

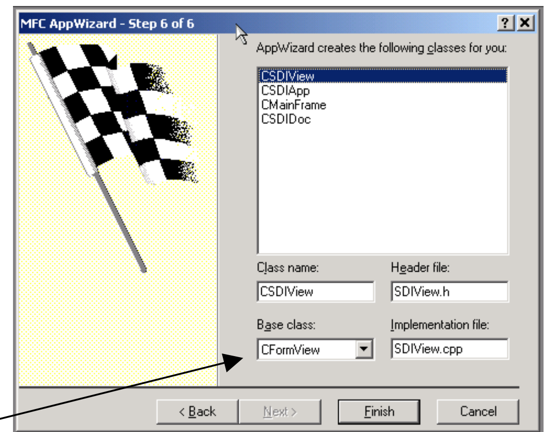
4 - MFC AppWizard Step 3/6 : laisser les options par défaut (l'application ne sera pas conteneur d'ActiveX mais pourra exploiter des contrôles ActiveX)

5 - MFC AppWizard Step 4/6 : choix d'options de l'application

Cocher -> Barre d'outils "dockable"[2]
Cocher -> Barre d'état (sous la fenêtre de vue)
Ne pas cocher -> Commandes Print/Print Preview/Print Setup

[2] une barre "dockable" est une fenêtre avec des boutons que l'utilisateur peut ancrer ou détacher de la fenêtre cadre.

6 - MFC AppWizard Step 5/6 : laisser les options par défaut



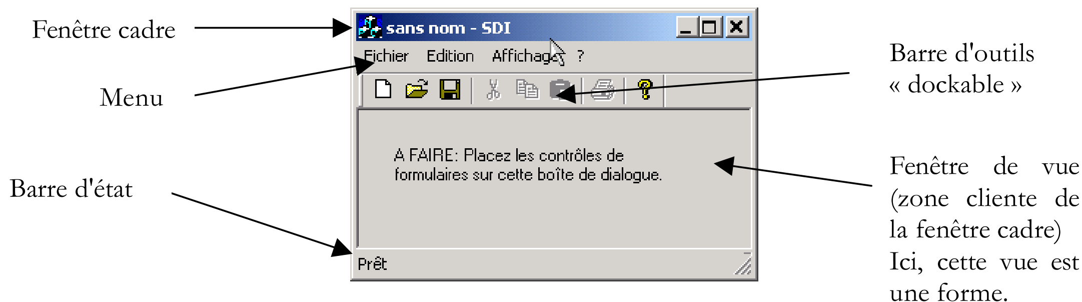
7 - MFC AppWizard Step 6/6 : **choix de la classe de vue**

L'étape 6 résume les classes qui vont être créées par le générateur d'application pour ce modèle SDI, il s'agit des classes :

```
CSDIView (classe de vue)
CSDIApp (classe d'application)
CMainFrame (classe de fenêtre cadre)
CSDIDoc (classe de document)
```

A cette étape, on peut choisir la classe de base de la classe de vue : sélectionner la classe **`CFormView`** (on souhaite que la vue soit une fome).

Compiler/linker et exécuter l'application pour obtenir l'application décrite ci-dessous.



10.3 Rôle des classes du modèle SDI

L'assistant `AppWizard` crée le projet SDI (Single Document Interface) initial à partir des différentes informations obtenues durant les 6 étapes de l'assistant. Lorsque l'architecture *Document/Vue* est utilisée dans l'application (ce qui est recommandé pour faciliter la sauvegarde du document sur fichier via la sérialisation), l'application est répartie dans les classes suivantes (on examine ici le cas où le projet est nommé **SDI**).

La classe d'application (`CSDIApp`)

Cette classe correspond au thread principal de l'application Windows. La classe **`CSDIApp`** dérive de **`CWinApp`** (qui elle même dérive de **`CWinThread`**). Un objet global de cette classe d'application est créé dans le fichier **`SDI.cpp`**

```
CSDIApp theApp;
```

et la méthode virtuelle **`CSDIApp::InitInstance()`** est appelée sur cet objet, depuis la fonction `AfxWinMain()`, pour initialiser l'application. On rappelle que la méthode **`CSDIApp::InitInstance()`** joue le rôle de point d'entrée d'une application Windows.

L'architecture Document/Vue(s) (variante du design pattern *Observer*)

Lorsque l'on coche l'option **`Architecture Document/View`** dans la première fenêtre de l'assistant `AppWizard`, on indique au générateur de projet qu'on souhaite utiliser le modèle ***Document/Vue*** qui est une variante du modèle de conception ***Sujet/Observateurs*** présenté dans la section 8 (application avec architecture MVC). Dans ce cas, l'application suit une architecture MVC (Modèle-Vue-Contrôleur) : le ***Modèle*** (M) est géré par un objet ***document*** de type **`CSDIDoc`** (classe dérivée de **`CDocument`**) et les parties ***Vue et Contrôleur*** (VC) sont contenues dans un (ou des) objet(s) de type **`CSDIView`** (classe dérivée de **`CView`**).

On rappelle que les fenêtres Windows contiennent souvent à la fois une partie contrôleur (entrées utilisateurs) et une partie vue (affichage de données), ces deux parties sont donc gérées au sein de mêmes objets "fenêtres de vue".

Par rapport au design pattern *Observer* (ou modèle de conception Sujet/Observateurs) développé dans la section 8, l'objet de type **`CDocument`** (ou dérivé) joue le rôle du *sujet,* et le(s) objet(s) de type **`CView`** (ou dérivé) joue(nt) le rôle d'*observateur(s) du document.*

La classe de document (`CSDIDoc`) : la partie modèle de l'architecure MVC

Un objet document contient la partie Modèle d'une application MVC. Le document est donc prévu pour contenir les classes métier, c'est-à-dire les données visualisables et modifiables par l'utilisateur. En outre, le document dispose d'une méthode pour notifier ses vues (objets dérivés de **`CView`**) lorsque son état change, il s'agit de la méthode **`CDocument::UpdateAllViews()`** définie dans le fichier source `DOCCORE.CPP`

```
void CDocument::UpdateAllViews(CView* pSender, LPARAM lHint, CObject* pHint){
        ASSERT(pSender == NULL || !m_viewList.IsEmpty());
        POSITION pos = GetFirstViewPosition();
        while (pos != NULL)    {
                CView* pView = GetNextView(pos);
                ASSERT_VALID(pView);
                if (pView != pSender) pView->OnUpdate(pSender, lHint, pHint);
        }
}
```

De plus, comme la classe `Sujet` de la section 8, la classe `CDocument` contient des méthodes pour attacher/détacher des vues (le code ci-dessous vient également du fichier `DOCCORE.CPP`)

```
// méthode qui attache une vue au document (càd un observateur à son sujet)
void CDocument::AddView(CView* pView)
{
        ASSERT_VALID(pView);
        ASSERT(pView->m_pDocument == NULL);  // must not be already attached
        ASSERT(m_viewList.Find(pView, NULL) == NULL);   // must not be in list
        m_viewList.AddTail(pView);
        ASSERT(pView->m_pDocument == NULL);  // must be un-attached

        /* la vue (l'observateur) contient un pointeur sur son document (le sujet)*/
        pView->m_pDocument = this;

        OnChangedViewList();    // must be the last thing done to the document
}

// méthode qui détache une vue du document
void CDocument::RemoveView(CView* pView)
{
        ASSERT_VALID(pView);
        ASSERT(pView->m_pDocument == this); // must be attached to us
        m_viewList.RemoveAt(m_viewList.Find(pView));
        pView->m_pDocument = NULL;
        OnChangedViewList();    // must be the last thing done to the document
}
```

La sérialisation du document : enregistrement du document sur fichier (voir aussi la section 9)

La classe de document **`CSDIDoc`** (dérivée de `CDocument`) est définie comme suit (des parties ont été supprimées)

```
class CSDIDoc : public CDocument
{
protected:
        CSDIDoc();
        DECLARE_DYNCREATE(CSDIDoc)
        //{{AFX_VIRTUAL(CSDIDoc)
        public:
        virtual BOOL OnNewDocument();
        virtual void Serialize(CArchive& ar);         // sérialisation du document
        //}}AFX_VIRTUAL
public:
        virtual ~CSDIDoc();
protected:
        DECLARE_MESSAGE_MAP()
};
```

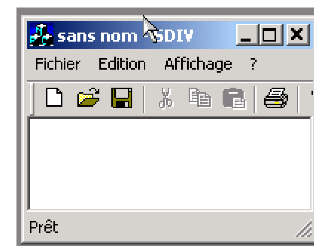
On voit ci-dessus que le mécanisme de sérialisation du document est prévu. On voit en effet que la méthode **`CSDIDoc::Serialize()`** est définie dans la classe **`CSDIDoc`**. Le traitement permettant de choisir un fichier puis de lire ou sauvegarder le document dans le fichier choisi suite aux commandes `Fichier|Ouvrir` et `Fichier|Enregistrer` est déjà implémenté. Dans ce mécanisme pré-programmé, il reste seulement à indiquer dans la méthode `CSDIDoc::Serialize(CArchive & ar)` quelles sont les données à lire/sauvegarger.

`CSDIDoc::Serialize(CArchive & ar)` : méthode où l'on doit programmer la sérialisation des classes métier

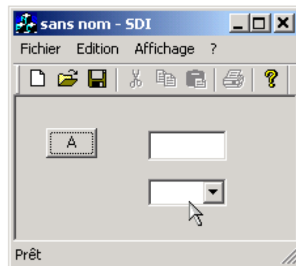
La classe de vue (`CSDIView`)

Dans l'architecture `Document/View` des MFC, le document fournit ses données à une ou plusieurs vues (objets dérivés de `CView`). En pratique, les objets vue de type `CView` ou dérivé, sont des objets gérant des fenêtres Windows. Pour les applications SDI, la fenêtre de vue correspond à la zone cliente de la fenêtre cadre.

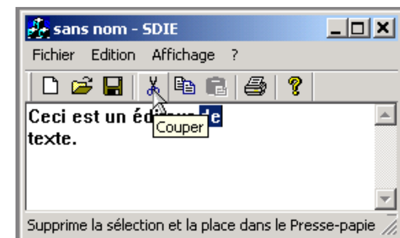
Avec les MFC, on peut utiliser différentes classes de base pour les vues. La classe de base de la classe de vue (choisie à l'étape 6 de `AppWizard`) détermine l'apparence de la vue.



Si l'on choisit **`CView`** comme classe de base pour la classe de vue, on obtient l'application SDI ci-contre. La vue est une zone blanche dans laquelle il est possible de réaliser notamment du dessin.



Si l'on choisit **`CFormView`** comme classe de base pour la classe de vue, on obtient l'application SDI ci-contre. La vue est une **forme** dont l'apparence est décrite par une ressource (comme pour les boîtes de dialogue). Une vue de type **`CFormView`** peut héberger des fenêtres de contrôle (bouton, édit box, combo box ...) et des contrôles ActiveX.



Si l'on choisit **`CEditView`** comme classe de base pour la classe de vue, on obtient l'application SDI ci-contre. La vue est une zone permettant de saisir du texte. Toutes les fonctionnalités telles que copier/coller, Fichier|Enregistrer sont déja implémentées. En résumé, l'application obtenue est un éditeur de texte que l'on peut faire évoluer.

MCours.com

Toutes les classes de vue MFC dérivent de CView. On retrouve donc dans cette classe les méthodes permettant de dialoguer avec le document (revoir la classe Observateur du modèle de conception Sujet/Observateurs).

Une classe de vue dispose notamment d'un pointeur **m_pDocument** sur le document qui est initialisé par l'objet document quand une vue lui est attachée (revoir le code source de CDocument::AddView()), et d'une méthode virtuelle **CView::OnUpdate()** qui est appelée par le document pour informer la vue que son état a changé.

```
class CView : public CWnd
{
protected:
       CView();
public:
       CDocument* GetDocument() const;             // retourne m_pDocument

       /**** méthode virtuelle appelée par le document *****/
       virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
protected:
       CDocument* m_pDocument;      // pointeur sur le document
       ...
};
```

La classe de fenêtre cadre (CMainFrame)

La classe **CMainFrame** dérive de CFrameWnd . La fenêtre cadre (fenêtre principale de l'application SDI) est gérée à travers une instance de la classe **CMainFrame**. On voit en outre que les barres d'outils et d'état sont gérées à travers deux objets membres de la fenêtre cadre.

```
class CMainFrame : public CFrameWnd
{
protected:                   // create from serialization only
       CMainFrame();
       DECLARE_DYNCREATE(CMainFrame)
       // . . .
public:
       virtual ~CMainFrame();
protected:           // control bar embedded members
       CStatusBar  m_wndStatusBar;   // objet gérant la barre d'état
       CToolBar    m_wndToolBar;            // objet gérant la barre d'outils
};
```

10.4 Séquence de création des objets d'une application SDI

Pour les applications SDI avec architecture Document/View, la méthode **C__App::InitInstance()** contient la création d'un modèle de document (doc template). Ce modèle de document se charge de créer la fenêtre cadre (ici, une instance de CMainFrame), le document (une instance de CSDIDoc) et la vue (une instance de CSDIView). De plus, le modèle de document gére les liens entre fenêtre cadre, document et vue(s). Ainsi, pour notre application SDI (le projet s'appelle aussi SDI) on obtient :

```
BOOL CSDIApp ::InitInstance()
{
       ...
       CSingleDocTemplate* pDocTemplate;
       pDocTemplate = new CSingleDocTemplate(
              IDR_MAINFRAME,                    // ID resource Menu/Toolbar
              RUNTIME_CLASS(CSDIDoc),           // document class
              RUNTIME_CLASS(CMainFrame),        // main SDI frame window
              RUNTIME_CLASS(CSDIView));         // view class
       AddDocTemplate(pDocTemplate);
       ...
}
```

Le mécanisme utilisé ici pour pouvoir créer des instances des différentes classes (fenêtres cadre, vue et document) est propre à MFC, il repose sur la notion de métaclasse (CRunTimeClass) permettant d'identifier une classe.

Tout d'abord, il faut noter que les classes MFC n'utilisent pas le mécanisme RTTI (Run Time Type Information) du C++ pour identifier les types à l'exécution (notamment parce que le mécanisme RTTI n'était pas disponible de manière standard lors des premières versions des MFC). Les MFC implémentent une reconnaissance de type à l'aide de méthodes virtuelles. Dans une classe **CLS** dérivée de la classe Base (et ayant la classe CObject dans son asendance), les macros **DECLARE_DYNAMIC(CLS)** et **IMPLEMENT_DYNAMIC(CLS,Base)** permettent d'ajouter à la classe **CLS** un objet membre statique public **CLS::classCLS**. Cet objet est une instance de la métaclasse **CRuntimeClass** (cet objet permet d'identifier la classe) et une méthode virtuelle **CRuntimeClass* CLS::GetRuntimeClass() const** retournant l'adresse de l'objet **CLS::classCLS** identifiant la classe. La classe **CRuntimeClass** est dite *métaclasse* car une instance de cette classe permet d'identifier une autre classe.

En plus de la reconnaissance de type via un objet de type CRunTimeClass, les classes CMainFrame, CSDIView et CSDIDoc utilisent la possibilité de créer des objets dynamiques (des objets alloués dans le tas) en passant par une méthode statique. En effet, on remarque que les constructeurs de ces classes sont à accès protégé. En d'autres termes, l'utilisateur de ces classes ne peut pas créer d'instance directement, il doit donc passer par une méthode statique ajoutée à chacune des classes (revoir le modèle de classe Singleton de la section 8).

Pour une classe CLS dérivée de BaseClass et ayant CObject dans son ascendance (d'ailleurs BaseClass peut être CObject), l'appel de la macro **DECLARE_DYNCREATE(CLS)** dans la définition de la classe CLS (fichier header CLS.h), ainsi que l'appel de la macro **IMPLEMENT_DYNCREATE(CLS,BaseClass)** dans le fichier d'implémentation CLS.cpp permettent d'ajouter la méthode statique

```
CObject * CLS::CreateObject()
```

qui retourne l'adresse d'un nouvel objet de type CLS alloué dans le tas. Logiquement, on voit donc dans les classes (fichier header et fichier d'implémentation) de fenêtre cadre, de vue et de document les appels des macros ci-dessous

```
DECLARE_DYNCREATE(CSDIDoc),
IMPLEMENT_DYNCREATE(CSDIDoc, CDocument),

DECLARE_DYNCREATE(CMainFrame),
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

DECLARE_DYNCREATE(CSDIView),
IMPLEMENT_DYNCREATE(CSDIView,CFormView)
```

Grâce à ces macros, ces classes ont désormais la possibilité de créer un objet dynamique (même si les constructeurs sont protégés). On peut en outre créer un objet de ces classes en connaissant l'objet CRuntimeClass qui les identifie. Par exemple, le code ci-dessous alloue un objet de la classe CSDIView.

```
/* la macro RUNTIME_CLASS(CSDIView) fournit l'adresse de
l'objet CSDIView::classCSDIView de type CRuntimeClass */
CRuntimeClass * pRTC=RUNTIME_CLASS(CSDIView);

// permet d'appeler, via pRTC, la méthode statique CSDIVIew::CreateObject()
pRTC->CreateObject();
```

Finalement, dans la méthode CSDIApp::InitInstance(), le constructeur de l'objet **CSingleDocTemplate** reçoit en paramètre les adresses des objets de type CRuntimeClass permettant d'identifier les classes de document, de fenêtre cadre et de fenêtre de vue. Grâce à ces objets de type CRunTimClass, l'objet CSingleDocTemplate est capable de créer des instances des classes CMainFrame, CSDIView et CSDIDoc et de créer les liens entre ces objets.

10.5 Changer la classe de vue d'une application SDI

L'assistant `AppWizard` permet à l'étape 6 de choisir entre différents types de vue :

`CView` , `CFormView` (vue pour gérer des contrôles), `CEditView` (éditeur de texte), `CHtmlView`...

Supposons que le classe vue choisie initialement ne convienne plus après quelques développements. Il reste alors possible de modifier ***a posteriori*** la classe de vue. En principe, si l'on respecte l'architecture Document/Vue qui découple la partie métier de la partie visualisation, le changement de vue n'implique pas de gros changements dans le logiciel. Pour changer la classe de vue, il faut

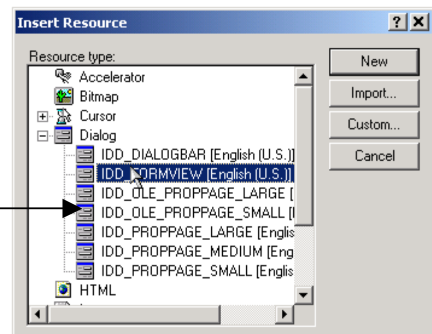
- créer une nouvelle classe dérivée de **`CView`**
- remplacer la classe de vue au moment de la création du doc template dans `C__App ::InitInstance()`
- supprimer les références à la classe de vue précédente

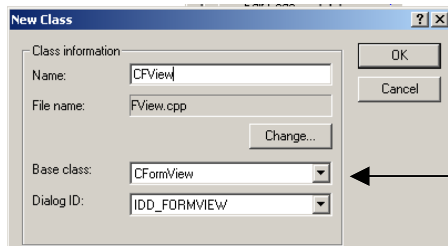
Exemple : créer une application SDI (nom de projet **`SDI`**) avec l'architecture Document/View et choisir **`CView`** comme classe de base pour la vue (à l'étape 6/6 de l'assistant `AppWizard`). Compiler et exécuter cette application.

L'objectif est de remplacer cette classe de vue par une vue dérivée de **`CFormView`.** Ajouter une nouvelle classe de vue **`CFView`** (une forme) au projet.

1) Ajouter une ressource pour la forme :

`Insert |Resource | Dialog | IDD_FORMVIEW`





2) utiliser `ClassWizard` pour créer la classe **`CFView`** associée

sélectionner **`CFormView`** comme classe de base

3) modifier la création du modèle de document dans **`CSDIApp::InitInstance()`**

```
#include "FView.h"        // insertion de la déclaration de la classe CFView

BOOL CSDIApp::InitInstance()
{
                ...

        CSingleDocTemplate* pDocTemplate;
        pDocTemplate = new CSingleDocTemplate(
                IDR_MAINFRAME,
                RUNTIME_CLASS(C___Doc),
                RUNTIME_CLASS(CMainFrame),        // main SDI frame window
                //RUNTIME_CLASS(CView));          // ancienne vue
                RUNTIME_CLASS(CFView));           // nouvelle vue de type CFView
        AddDocTemplate(pDocTemplate);

                ...
}
```

Remarque : si la classe de vue précédente n'est plus utilisée, les fichiers `.h` et `.cpp` de cette classe peuvent être supprimés du projet et effacés du répertoire du projet.

10.6 Exploiter une classe métier dans une application SDI avec l'architecture Document/View.

Dans la section 8, on a développé de bout en bout une application sur le modèle MVC. On peut reprendre ici les classes métier **`InterfaceRepertoire`**, **`Repertoire`** et **`Fiche`** de cette application pour les exploiter au sein d'une application SDI MFC.

Pour réutiliser ces classes dans une application MFC avec l'architecture Document/View, il faut néanmoins supprimer la relation hiérarchique « `InterfaceRepertoire` dérive de `Sujet` », puisque c'est ici le document qui va donner accès à l'objet `Repertoire`. On préfère donc ici placer dans la classe de document `CSDIDoc` un pointeur de type `InterfaceRepertoire*` capable de stocker l'adresse de l'instance unique de la classe `Repertoire`.

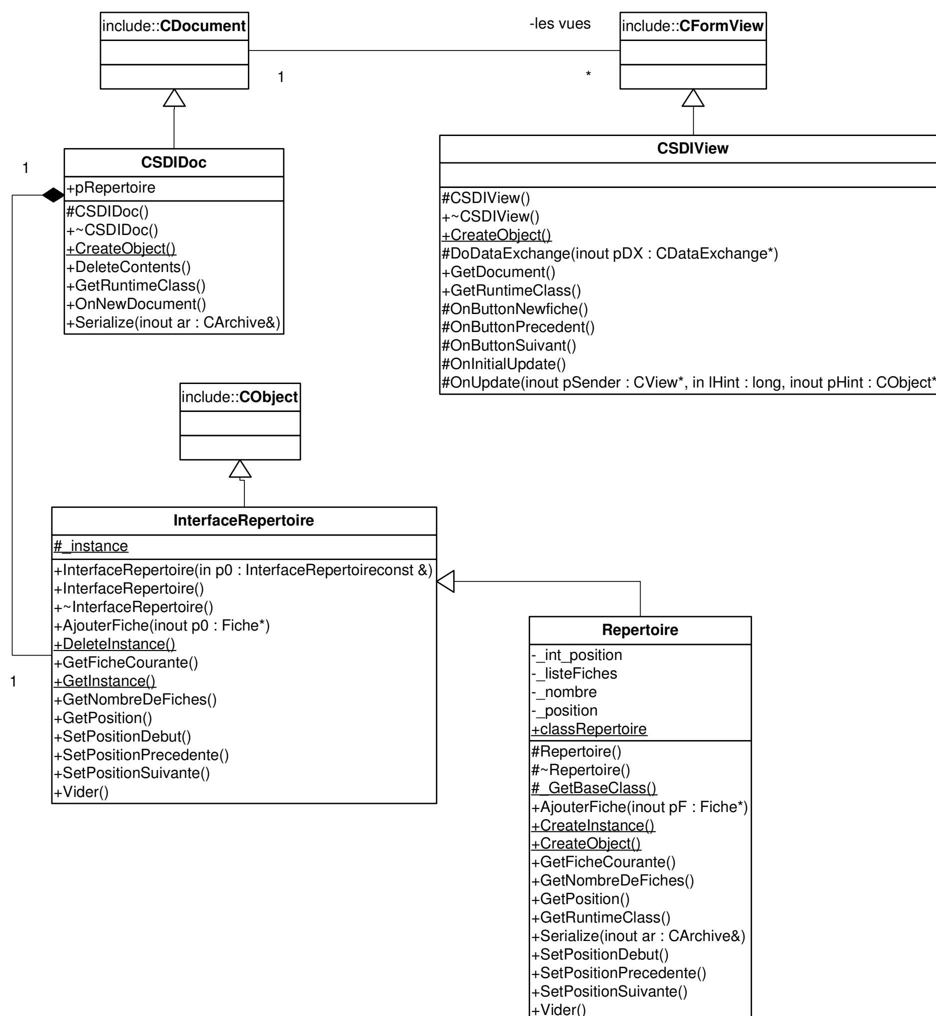
Le diagramme de classes ci-dessous décrit les relations entre les classes de l'application SDI[3]. On rappelle que la classe `Repertoire` sait "se sérialiser" dans une archive. Les points importants sont les suivants.

`InterfaceRepertoire * CSDIDoc::m_pRepertoire` = pointeur sur l'objet `Repertoire`

`void CSDIDoc::Serialize(CArchive &ar)` = appel de la sérialisation de l'objet `Repertoire`

`void CSDIView::OnUpdate( /*…*/ )` = méthode de mise à jour de la vue à partir des données du document

`CSDIDoc * CSDIView::GetDocument()` = méthode retournant l'adresse de l'objet document



Compléter l'application SDI suivant les spécifications suivantes

---
[3] le diagramme de classes a été fait par rétroconception Microsoft Visual C++ vers Microsoft Visio

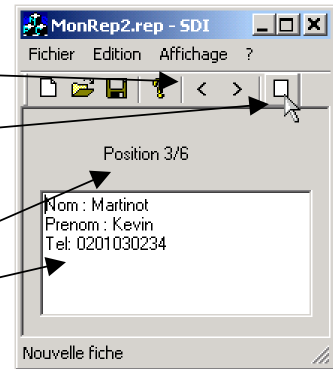
- la navigation dans le répertoire se fait depuis la barre d'outils

bouton < fiche précédente
bouton > fiche suivante

- l'ajout d'une nouvelle fiche se fait en cliquant sur un bouton de la barre d'outils

- la sérialisation du document est possible (depuis le menu `Fichier` ou depuis les boutons de la barre d'outils)

Affichage dans la vue : position courante et fiche courante



Procédure

1) Créer avec `AppWizard` une application **`MFC SDI Document/View`**

Nom de Projet : **SDI**

2) Créer dans le répertoire **`SDI`** du projet, un sous-répertoire `Metier` dans lequel on place les classes métier

`InterfaceRepertoire.h InterfaceRepertoire.cpp` : classe abstraite

`Repertoire.h Repertoire.cpp` : classe concrète de répertoire

`Fiche.h Fiche.cpp` : classe de fiches

3) Compléter le fichier **`"StdAfx.h"`** par les inclusions des fichiers headers des classes métier

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
#if !defined(AFX_STDAFX_H__83B518AF_6664_4ECC_BE75_D9B28786BEF9__INCLUDED_)
#define AFX_STDAFX_H__83B518AF_6664_4ECC_BE75_D9B28786BEF9__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#define VC_EXTRALEAN         // Exclude rarely-used stuff from Windows headers
#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC Automation classes
#include <afxdtctl.h>        // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>                 // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT

#include "Metier\InterfaceRepertoire.h"
#include "Metier\Repertoire.h"

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.
#endif // !defined(AFX_STDAFX_H__83B518AF_6664_4ECC_BE75_D9B28786BEF9__INCLUDED_)
```

4) Dans le navigateur **`File View`**, ajouter les fichiers header (.h) et d'implémentation (.cpp) des classes métier au projet

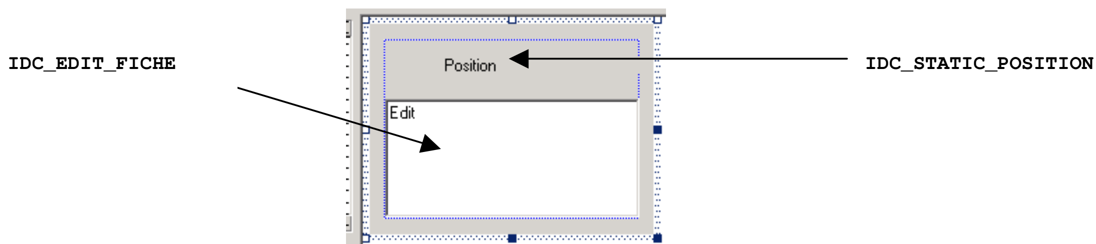
5) Ajouter un pointeur **public** **CSDIDoc::m_pRepertoire** de type **InterfaceRepertoire *** à la classe **CSDIDoc**. Ce pointeur va donner accès à l'instance unique du répertoire, via l'objet document, aux différentes vues. Normalement, puisque l'inclusion du fichier **InterfaceRepertoire.h** est faite dans le fichier **StdAfx.h**, il n'est pas nécessaire de remettre l'inclusion du fichier `InterfaceRepertoire.h` dans le fichier `SDIDoc.h`

6) Ajouter la création de l'instance de `Repertoire` dans le constructeur de **CSDIDoc** et la destruction de cette instance dans le destructeur.

```
----------------------------------------------------------------
// SDIDoc.h
// #include "Metier\InterfaceRepertoire.h"  // normalement inutile

class CSDIDoc : public CDocument{
       // ...
public:
              InterfaceRepertoire * m_pRepertoire; // pour que les vues y accèdent
              virtual ~CSDIDoc();
};
----------------------------------------------------------------
----------------------------------------------------------------
// SDIDoc.cpp (extrait)
CSDIDoc::CSDIDoc(){
       Repertoire::CreateInstance(); // crée l'instance du répertoire
       m_pRepertoire=InterfaceRepertoire::GetInstance();
}

CSDIDoc::~CSDIDoc(){
       InterfaceRepertoire::DeleteInstance();      //libère l'instance
}

void CSDIDoc::Serialize(CArchive& ar){
       m_pRepertoire->Serialize(ar); //sérialisation du répertoire
}

/* méthode utilisée lorsque l'utilisateur demande à créer un nouveau document. L'objet document
courant est conservé mais réinitialisé par cette méthode */
void CSDIDoc::DeleteContents(){
       m_pRepertoire->Vider();       // méthode supprimant les fiches
       CDocument::DeleteContents();
}
       . . .
----------------------------------------------------------------
```

7) Editer la ressource de la forme et associer des variables aux contrôles

CSDIView::m_edit_fiche : variable `CString` associée à **IDC_EDIT_FICHE**
CSDIView::m_position_str : variable `CString` associée à **IDC_STATIC_POSITION**



8) Compléter la classe de vue :

- Ajouter la méthode **OnUpdate()** qui est appelée par le document pour mettre à jour la vue
- Ajouter les gestionnaires d'événements : click sur le bouton < , click sur le bouton > , click sur le bouton nouvelle fiche

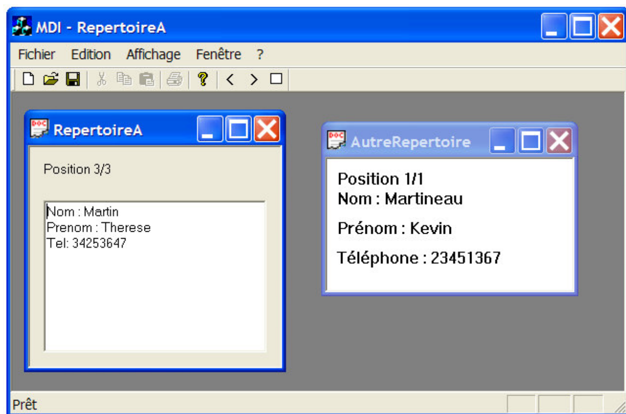
```
---------------------------------------------------------------
// SDIView.cpp (extraits)

// Equivaut à la méthode Update de la classe Observateur
void CSDIView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
	/* mise à jour de la vue lors du changement du document */
	Fiche * pF=GetDocument()->m_pRepertoire->GetFicheCourante();
	if(pF!=NULL) pF->Dump(m_edit_fiche);
	else m_edit_fiche.Empty();
	m_position_str.Format("Position%d/%d",
				GetDocument()->m_pRepertoire->GetPosition()+1,
				GetDocument()->m_pRepertoire->GetNombreDeFiches());
	UpdateData(FALSE);
}

// gestionnaire de click sur le bouton précédent
void CSDIView::OnButtonPrecedent(){
	GetDocument()->m_pRepertoire->SetPositionPrecedente();
	GetDocument()->UpdateAllViews(this); //demande la mise à jour des vues
	OnUpdate(this,0,NULL); // la vue courante ne va pas recevoir le msg OnUpdate
}

// gestionnaire de click sur le bouton suivant
void CSDIView::OnButtonSuivant(){
	GetDocument()->m_pRepertoire->SetPositionSuivante();
	GetDocument()->UpdateAllViews(this);
	OnUpdate(this,0,NULL); // la vue courante ne va pas recevoir le msg OnUpdate
}

// gestionnaire de click sur le bouton nouvelle fiche
void CSDIView::OnButtonNewfiche(){
	CNouvelleFicheDlg dlg;
	if(dlg.DoModal()==IDOK)
	{
		Fiche * pF= Fiche::AllouerObjet(dlg.m_nom,dlg.m_prenom,dlg.m_telephone);
		GetDocument()->m_pRepertoire->AjouterFiche(pF);
		GetDocument()->UpdateAllViews(this);
		OnUpdate(this,0,NULL);

/* Indiquer que le document a été modifié. Ainsi, un message demandera à l'utilisateur s'il
souhaite enregistrer le document avant de fermer l'application */

		GetDocument()->SetModifiedFlag();
	}
}
---------------------------------------------------------------
```

10.7 Les applications MDI (Muli Document Interface)

L'assistant AppWizard de génération du projet laisse le choix entre 3 types d'applications : Dialog Based, SDI et MDI. Les deux premiers modèles ont été étudiés précédemment. On a vu que dans le modèle SDI, l'architecture repose sur une architecture MVC. La classe dérivée de **CDocument** contient les références aux objets métier.

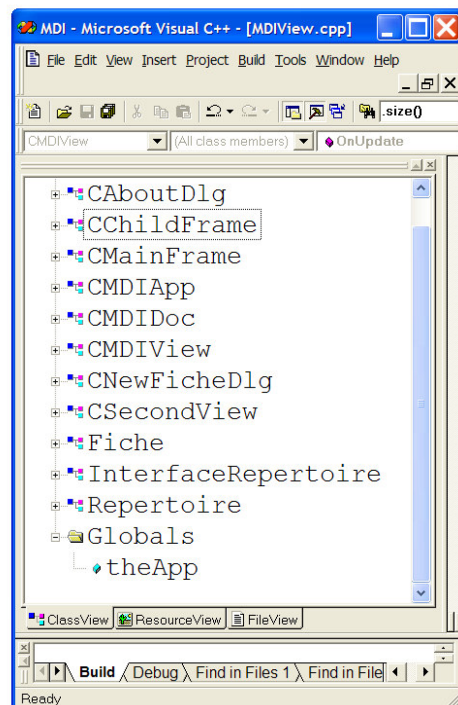
Dans une application SDI, l'utilisateur ne peut accéder qu'à un seul document, même si plusieurs vues peuvent y être connectées. Comme leur nom l'indique, les applications MDI permettent à un utilisateur d'accéder simultanément à différents documents. Il s'agit le plus souvent de documents réalisés à partir de la même classe de document. Pour les applications MDI, la fenêtre cadre principale gère une fenêtre cadre fille par document. Chaque fenêtre cadre fille contient une vue (ou plusieurs vues) donnant accès à un document.

L'exemple ci-contre présente la version MDI de la gestion de répertoires de numéros de téléphone. Cette fois-ci, le répertoire ne doit plus être géré par une classe singleton car plusieurs objets répertoire sont nécessaires.



Dans l'exemple ci-contre, chaque document créé peut être visualisé par une vue différente. La fenêtre de gauche visualise la fiche courante du document `RepertoireA` (c'est le nom de fichier) dans une forme, alors que la fenêtre de droite visualise la fiche courante du document `AutreRepertoire` dans une vue dérivée de **`CView`**.

Pour les applications MDI, la relation entre le document et ses vues reste la même. En revanche, chaque document utilise une fenêtre cadre fille différente. Le projet contient donc une classe de fenêtre cadre principale `CMainFrame` et une classe de fenêtre cadre fille `CChildFrame`. La vue d'un document est hébergée par une fenêtre cadre fille. L'application (MDI) complète utilise donc les classes suivantes :



`CMDIApp` : classe d'application
`CMainFrame` : classe de la fenêtre cadre principale
`CChildFrame` : classe de fenêtre cadre fille
`CMDIDoc` : classe de document
`CMDIView` : classe de vue (dérive de `CFormView`)
`CSecondView` : seconde classe de vue (dérive de `CView`)
`CNewFicheDlg` : classe de boîte de saisie d'une fiche

Les classes metier `Fiche`, `InterfaceRepertoire`, `Repertoire`

Pour cette application, deux modèles de document sont créés

```
BOOL CMDIApp::InitInstance(){
// . . .
        CMultiDocTemplate* pDocTemplate;
        pDocTemplate = new CMultiDocTemplate(
                IDR_MDITYPE,
                RUNTIME_CLASS(CMDIDoc),
                RUNTIME_CLASS(CChildFrame),
                RUNTIME_CLASS(CMDIView));   //vue forme
        AddDocTemplate(pDocTemplate);
        pDocTemplate = new CMultiDocTemplate(
                IDR_MDITYPE2,
                RUNTIME_CLASS(CMDIDoc),
                RUNTIME_CLASS(CChildFrame), // custom MDI child frame
                RUNTIME_CLASS(CSecondView)); // vue CView
        AddDocTemplate(pDocTemplate);
// . . .
```

Lorsque l'on crée une application MDI avec un seul type de document et un seul type de vue par document, il y a peu de différences de développement avec le modèle SDI, puisque là encore le programmeur n'a pas à se soucier de la création des documents et de la façon dont ceux-ci sont attachés aux vues.

Les cas où différents types de documents sont gérés, voire différents types de vue, sont moins évidents à développer. Il est alors nécessaire de trouver des exemples qui expliquent le rôle du modèle de document (doc template).

11 Programmation réseau (les sockets)

Cette section aborde les techniques de programmation d'applications réseau type Client-Serveur en utilisant la classe `CAsyncSocket` qui encapsule l'API Socket.

11.1 Le modèle client-serveur, interface sockets

La communication au niveau des applications s'effectue suivant un modèle très répandu, nommé Client-Serveur. L'application qui établit le contact s'appelle le *client* alors que celle qui attend s'appelle le *serveur*. Pour interfacer une application et les protocoles de communication on utilise l'API (Application Program Interface) **Socket**, issue du système UNIX BSD, mais aujourd'hui présente sur de nombreux systèmes d'exploitations. Un socket représente une extrémité du canal de communication. De nombreux documents sont présents en ligne, vous trouverez un cours et une description de l'utilisation des fonctions de l'API socket en C sur la machine lisant `R:\partage\tpreseau\socket_program`.

La bibliothèque MFC propose deux classes qui encapsulent l'API Socket. Il s'agit des classes `CAsyncSocket` et `CSocket`. Celà signifie que les fonctions de l'API Socket vont alors être accessibles sous la forme de méthodes attachées à une classe C++.

Dans ce document, seule la classe **`CAsyncSocket`** est présentée. Celle-ci donne accès, sous forme d'une classe C++, au comportement de bas niveau des sockets (écoute, connexion, envoi et réception de données ...). Il faut savoir que les MFC proposent une seconde classe `CSocket` offrant une abstraction plus importante, et permettant l'échange de données (via le réseau) en utilisant l'interface `CArchive`. L'utilisation de la classe `CSocket` ne fait donc pas apparaître toutes les techniques de bas niveau liées à l'utilisation de l'API Socket.

11.2 La classe CAsyncSocket

La classe `CAsyncSocket` s'utilise principalement par dérivation. En effet, la gestion des sockets par cette classe est une gestion asynchrone, ce qui signifie que les méthodes de cette classe ne sont pas bloquantes. En pratique, sur les systèmes Windows multi-tâches, chaque socket exploite son propre thread de travail. Le socket devient donc indépendant de l'activité du programme principal. Par conséquent, différents événements déclenchés par le socket doivent être exploités pour contrôler les différentes étapes des opérations de connexion et d'échanges de données.

Qu'il s'agisse d'un socket client ou serveur, les différents événements liés à la vie du socket conduisent à l'appel de méthodes virtuelles de la classe `CAsyncSocket`. Il faut donc surdéfinir ces méthodes dans une classe dérivée pour pouvoir les spécialiser.

Méthodes virtuelles de notification (événements liés aux sockets)

Les méthodes de notification des sockets sont comparables aux gestionnaires d'événements. Ces méthodes sont appelées par la classe `CAsyncSocket` pour réagir aux différents événements du socket, et donc synchroniser celui-ci avec le programme qui l'exploite.

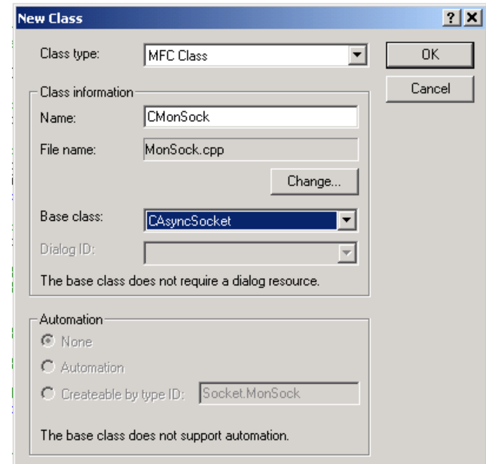
`CAsyncSocket::OnAccept()` : notifie un socket d'écoute (socket serveur) qu'il peut accepter une connexion

`CAsyncSocket::OnClose()` : notifie le socket courant que le socket situé à l'autre extrémité du canal de communication a fermé le canal.

`CAsyncSocket::OnConnect()` : notifie le socket client que le processus de connexion qu'il a demandé est terminé, un code d'erreur indique si la connexion est un **succès** ou un **échec**.

`CAsyncSocket::OnReceive()` : notifie un socket (client ou serveur) que des données ont été reçues et sont disponibles par la méthode `CAsyncSocket::Receive()`.

Créer une classe dérivée de **`CAsyncSocket`**



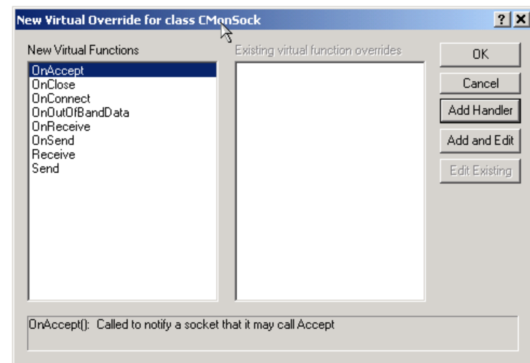
On peut parfois utiliser la classe `CAsyncSocket` directement, sans la spécialiser. Mais, il est toujours plus simple de la dériver pour pouvoir spécialiser les méthodes virtuelles de notification d'événements socket (présentés précédemment). En outre, l'IDE (Integrated Development Environment) Visual C++ simplifie la création d'une classe dérivée de `CAsyncSocket` grâce à un assistant de génération de classe.

La commande `Insert | New Class` lance l'assistant de création de classe.

class type : `MFC class`
name : *"nom de la classe dérivée"* (**`CMonSock`** sur l'exemple)
base class : **`CAsyncSocket`**

```
class CMonSock : public CAsyncSocket
{
public:
        CMonSock();
        virtual ~CMonSock();
public:
        //...
};
```

Ajouter une méthode virtuelle de notification à la classe dérivée de **`CAsyncSocket`**



Si vous créez les classes dérivées de `CAsyncSocket` avec l'assistant de création de classe, un autre assistant permet d'ajouter une méthode virtuelle de notification d'événement socket.

`Class View | Click droit sur la classe CMonSock | Add virtual function`

sélectionner la méthode virtuelle que vous voulez surdéfinir dans la classe dérivée, puis cliquer sur **`Add and Edit`.**

Instancier un objet de type `CAsyncSocket` et appeler sa méthode `Create` pour créer le descripteur SOCKET sous-jacent.

La création d'un socket respecte le schéma MFC d'une construction en deux étapes : instanciation d'un objet puis appel de la méthode `Create`.

Exemple :

```
CMonSock sock;        //objet automatique (le socket n'est pas créé)
sock.Create( );     // crée le socket en utilisant les paramètres par défaut

CMonSocket* pSocket = new CMonSock; // objet alloué dans le tas
int nPort = 27;
pSocket->Create(nPort);
```

Le premier objet **`CMonSock`** est un objet automatique réservé dans la pile. Le second objet est alloué dans le tas. Le premier appel de la méthode `Create` utilise les paramètres par défaut pour créer un socket flux. Le second appel crée un socket avec un numéro de port.

Il faut savoir que la durée de vie d'un socket géré par un objet de type `CAsyncSocket` va de l'appel de la méthode `Create` jusqu'à l'appel de la méthode `Close`. Le destructeur d'un objet `CAsyncSocket` ferme automatiquement le socket associé.

Les paramètres de la méthode `Create` sont les suivants (informations partiellement en anglais tirées de MSDN)

```
BOOL CAsyncSocket::Create(UINT nSocketPort = 0,
                          int nSocketType = SOCK_STREAM,
                          long lEvent = FD_READ | FD_WRITE
                          | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE,
                          LPCTSTR lpszSocketAddress = NULL );
```

Return Value : non nulle si la création du socket est possible; nulle sinon et un code d'erreur peut être obtenu par appel de la méthode int CAsyncSocket::GetLastError(); Les erreurs possibles (constantes entières) sont listées ci-dessous

WSANOTINITIALISED A successful AfxSocketInit must occur before using this API.
WSAENETDOWN The Windows Sockets implementation detected that the network subsystem failed.
WSAEAFNOSUPPORT The specified address family is not supported.
WSAEINPROGRESS A blocking Windows Sockets operation is in progress.
WSAEMFILE No more file descriptors are available.
WSAENOBUFS No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT The specified port is not supported.
WSAEPROTOTYPE The specified port is the wrong type for this socket.
WSAESOCKTNOSUPPORT The specified socket type is not supported in this address family.

nSocketPort
un numéro de port, ou 0 si l'on souhaite une sélection automatique du port.

Remarque : pour un socket serveur (socket d'écoute), vous devez spécifier un numéro de port. Pour un socket client, vous acceptez souvent la valeur par défaut de ce paramètre, qui laisse le socket sélectionner un port.

nSocketType
SOCK_STREAM (Utilise le protocole TCP) par défaut ou **SOCK_DGRAM** (Utilise le protocole UDP)

lEvent
Un masque logique qui indique les événements générés par l'objet socket (tous par défaut)

FD_READ Want to receive notification of readiness for reading.
FD_WRITE Want to receive notification of readiness for writing.
FD_OOB Want to receive notification of the arrival of out-of-band data.
FD_ACCEPT Want to receive notification of incoming connections.
FD_CONNECT Want to receive notification of completed connection.
FD_CLOSE Want to receive notification of socket closure.

lpszSockAddress
Un pointeur sur une chaîne de caractères contenant l'adresse d'un socket connecté, par exemple "128.56.22.8".

Destruction des objets CAsyncSocket

Si l'objet socket est créé sur la pile, son destructeur est appelé automatiquement lorsque l'objet passe hors de portée. La destruction de l'objet ferme le socket associé. Si l'objet socket est créé sur le tas, en utilisant l'opérateur **new**, celui-ci doit être désalloué avec l'opérateur **delete** pour que son destructeur soit appelé.

11.3 Chronologie des opérations Client-Serveur

La communication entre deux points d'un réseau, via des sockets, suit un schéma client-serveur. Côté serveur, un socket est créé et "écoute" sur un numéro de port donné. Ce socket attend qu'un socket distant (socket client situé en un autre point du réseau) demande une connexion.

L'application cliente crée un socket (socket client) et demande une connexion en précisant l'adresse et le numéro de port du serveur. Le socket d'écoute du serveur va détecter cette demande de connexion, accepter cette connexion et créer un socket de service pour assurer les échanges de données avec le client. A ce stade, tous les échanges de données se font entre le socket client (côté application cliente) et le socket de service du serveur (côté

application serveur). Pendant le temps où le serveur échange des données via un (ou plusieurs) socket de service, son socket d'écoute reste "à l'écoute" de nouvelles demandes de connexions de la part d'autres clients.

En résumé, l'application client gère un socket et l'application serveur crée un socket d'écoute et accepte les connexions sur autant de sockets de service que de clients connectés. Les étapes sont résumées par la chronologie ci-dessous. L'application serveur utilise deux classes dérivées de `CAsyncSocket` et l'application client utilise une classe dérivée de `CAsyncSocket`

`CSocketEcoute` : classe générant l'objet socket d'écoute **`sockEcoute`**
`CSocketService` : classe générant les sockets de service (un objet par client connecté)
`CSocketClient` : classe générant le socket client **`sockClient`**

Application Serveur	Application Client

//numéro de port d'écoute
int nPort=2000;
sockEcoute.Create(nPort);
sockEcoute.Listen();	

	sockClient.Create();
	int nPort=2000;
	sockClient.Connect("172.19.81.140",nPort);

CSocketEcoute::OnAccept(int nErr){ CSocketService * psockServ; psockService=**new CSocketService;** **Accept(*psockServ);** /*mémoriser ce pointeur dans une collection */ AddService(psockServ); CAsyncSocket::OnAccept(nErr); }	

	void CSocketClient::OnConnect(int nErr){
	/* si nErr==0, la connection
	est un succès */
	CAsyncSocket::OnConnect(nErr);
	}

	int Buffer[3]={10,20,-1}; sockClient.Send(Buffer,3*sizeof(int));

void CServiceSock::OnReceive(int nError)
{
int Buffer[20];
int dataSize; //taille des données reçues
dataSize=Receive(Buffer,20*sizeof(int));
CAsyncSocket::OnReceive(nError);
}	

Socket Serveur

Si le socket est un serveur, instancier un objet socket avec un numéro de port donné et créer le socket en écoute (avec `CAsyncSocket::Listen`). Lorsqu'un socket client demande à se connecter, la méthode virtuelle `CAsyncSocket::OnAccept()` du socket d'écoute est invoquée.

Remarque La fonction membre **`CAsyncSocket::Accept`** admet comme paramètre une référence à un nouvel objet `CAsyncSocket` vide. Vous devez créer une instance avant d'appeler **`CAsyncSocket::Accept`**. Si cet objet socket passe hors de portée (par exemple pour un objet automatique), la connexion se ferme. Ne pas appeler la fonction **`CAsyncSocket::Create`** pour ce nouvel objet socket.

Socket Client

Si le socket est un client, instancier un objet socket, appeler sa méthode `Create` (sans argument) puis appeler sa méthode `Connect` en fournissant l'adresse IP et le numéro de port du serveur. La méthode virtuelle `CAsyncSocket::OnConnect()` du client est appelée pour lui indiquer que le processus de connexion est terminé. Il peut s'agir d'un échec de connexion si aucun serveur n'écoute.

11.4 Utilisation de l'API Socket avec les MFC

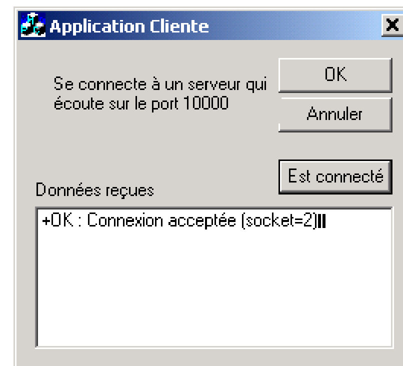
L'utilisation de la classe `CAsyncSocket` nécessite l'inclusion de fichier suivante (possible d'ajouter cette inclusion au fichier "`StdAfx.h`")

```
#include <afxsock.h>        // MFC socket extensions
```

En outre, l'API Socket doit être initialisée par l'appel de la fonction globale **`AfxSocketInit()`**. Cet appel est ajouté à la méthode `C__App::InitInstance()` lorsque l'on coche l'option `Windows Sockets` dans AppWizard.

```
BOOL CSocketApp::InitInstance()
{
      if (!AfxSocketInit()){
            AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
            return FALSE;
      }
      // ...
```

11.5 Exemple d'application Boîte de Dialogue avec utilisation d'un socket client



On donne ci-dessous des parties du code d'une application cliente (capture écran ci-dessous). Cette application boîte de dialogue se connecte à un serveur qui écoute sur le port 10000 et qui est exécuté sur la même machine que le client. On a créé une application `Dialog Based`, en cochant l'option Windows Sockets. On ajoute une classe `CMySock` dérivée de `CAsyncSocket`. On ajoute les méthodes `OnConnect`, `OnClose` et `OnReceive` ... à cette classe `CMySock`.

La classe de boîte de dialogue contient un objet membre de tpe `CMySock`. Le bouton de connexion déclenche la connexion au serveur. Si la connexion est acceptée, le texte du bouton est modifié. Le texte du bouton est également modifié lorsque le serveur ferme la connexion (méthode `CMySock::OnClose`).

L'objet socket connait l'objet boîte de dialogue grâce à la fonction globale `AfxGetMainWnd()`. En effet, ici, la fenêtre principale est la boîte de dialogue.

```
class CSockDlg; //déclaration anticipée

class CMySock : public CAsyncSocket
{
private:
	bool m_estConnecte;

public:
	CMySock();
	virtual ~CMySock();
	bool EstConnecte();
	CSockDlg * GetDlg();

//{{AFX_VIRTUAL(CMySock)
public:
virtual void OnConnect(int nErrorCode);
virtual void OnClose(int nErrorCode);
virtual void OnReceive(int nErrorCode);
//}}AFX_VIRTUAL
};
```

```
#include "MySock.h"
#include "sockdlg.h"

CMySock::CMySock(){
	m_estConnecte=false;
}

CMySock::~CMySock(){}

void CMySock::OnConnect(int nErrorCode)
{
  if(nErrorCode)
  {
    ::AfxMessageBox("Connexion échouée");
    Close();
  }
  else
  {
	m_estConnecte=true;
	GetDlg()->m_boutonC.SetWindowText("Est
	connecté");
  }
   CAsyncSocket::OnConnect(nErrorCode);
}

CSockDlg * CMySock::GetDlg(){
	return (CSockDlg*)::AfxGetMainWnd();
}

bool CMySock::EstConnecte(){
	return m_estConnecte;
}

void CMySock::OnClose(int nErrorCode){
  m_estConnecte=false;
  GetDlg()->m_boutonC.SetWindowText("Est fermé");
  CAsyncSocket::OnClose(nErrorCode);
}

void CMySock::OnReceive(int nErrorCode){
  char message[200];
  long taille = Receive(message,200);
  message[taille]=0;
  GetDlg()->m_listeDonnees.InsertString(-1,message);
  CAsyncSocket::OnReceive(nErrorCode);
}
```

On donne ci-dessous des extraits de la classe de boîte de dialogue. L'objet **`m_socket`** membre de la classe `CMySock` est créé et connecté au serveur par un click sur un bouton. La connexion s'effectue en local sur le port 10000.

```
void CSockDlg::OnButtonConnect(){
	if(!m_socket.EstConnecte())
	{
		m_socket.Create();
		m_socket.Connect("localhost",10000);
	}
}
```

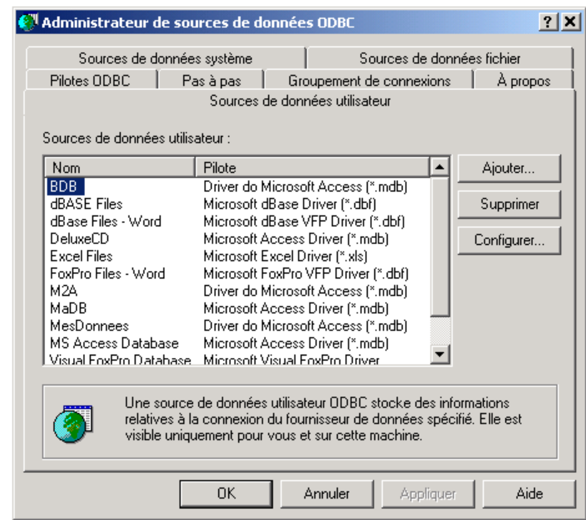
12 Accès aux bases de données avec ODBC

12.1 ODBC en deux mots

ODBC signifie *Open DataBase Connectivity*. Il s'agit d'un format défini par Microsoft permettant la communication entre des clients bases de données fonctionnant sous Windows et les SGBD du marché. Le gestionnaire ODBC est présent sur les systèmes Windows. Il existe toutefois des implémentations sur d'autres plates-formes, notamment des plates-formes Unix/Linux. Sous Windows, le gestionnaire ODBC est disponible dans le panneau de configuration/Outils d'administration/Sources de données ODBC.

La technologie ODBC permet d'interfacer de façon standard une application à n'importe quel serveur de bases de données, pour peu que celui-ci possède un driver ODBC (la quasi-totalité des SGBD du marché possèdent un tel pilote). Bien que ODBC permette un interfaçage avec des bases de données indépendamment du SGBD, cette

technologie reste une solution propriétaire de Microsoft. Cela se traduit par une dépendance de la plateforme (ODBC ne fonctionne que sur les plateformes Microsoft Windows). D'autre part, ODBC est fortement lié au langage C (utilisation de pointeurs), et ODBC utilise des paramètres non standards, ce qui le rend difficile à mettre en oeuvre directement dans les programmes.



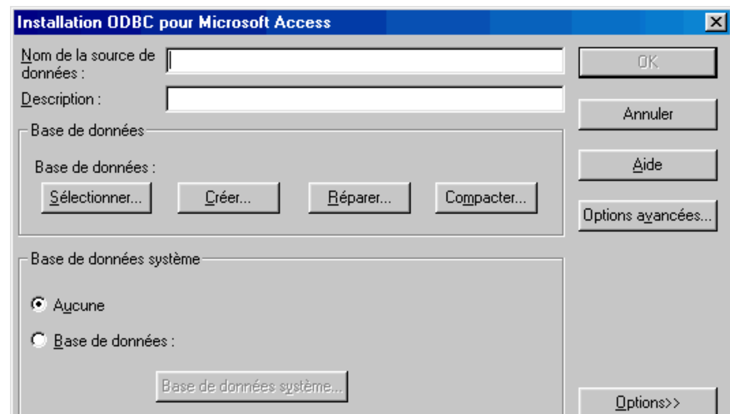
Qu'est-ce qu'un DSN ? ODBC permet de relier un client à une base de données en déclarant une source de données (correspondant généralement à une base de données) dans le gestionnaire ODBC (communément appelé *administrateur de source de données ODBC*). La source de données peut être aussi bien une base de données qu'un fichier Access, Excel ou bien même un fichier. On appelle donc *DSN* (*Data Source Name*) la déclaration de la source de données qui sera accessible par l'intermédiaire de ODBC.

Déclaration de la source de données L'administrateur de source de données ODBC, disponible dans le panneau de configuration (outils d'administration), permet de déclarer le type de données auxquelles il est possible d'accèder et de leur associer un nom. Pour déclarer une source de données il faut

- installer le driver ODBC pour la base de données si celle-ci n'est pas installée par défaut sous l'administrateur de source de données

- Etablir la liaison ODBC en ajoutant une source de données utilisateur

- L'administrateur de source de données va ensuite demander le nom à affecter à la source de données (il s'agit du nom par lequel la base de données sera accessible), va demander ensuite de sélectionner la source de données (un fichier dans le cas d'excel ou access, ou bien la base de données et éventuellement les tables à associer à la liaison ODBC).



- Il faut ensuite donner le chemin d'accès à la base de données en cliquant sur le bouton *sélectionner* de la fenêtre précédente

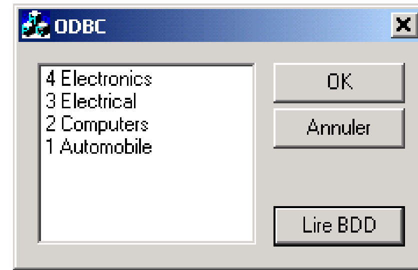
- La base de données devrait alors être accessible via ODBC

Suivant les bases de données, la procédure peut varier et des options supplémentaires peuvent-être demandées, mais la configuration d'un DSN reste globalement la même.

12.2 Les classes CDatabase et CRecordset

Un objet de la classe `CDatabase` gère une connexion à une base de données à travers un pilote ODBC. Ensuite, un objet `CRecordset` (associé à un objet `CDatabase`) va permettre la lecture/la mise à jour de la base. Ci-dessous, un exemple très simple de lecture d'une partie d'une table dans une base de base de données MS Access. La table `Categories` contient des enregistrements `CatID` et `Category`. Ces classes sont déclarées dans le fichier <afxdb.h>

CatID	Category
1	Automobile
2	Computers
3	Electrical
4	Electronics
5	Employment
6	Furniture




```
void CODBCDlg::OnLireBDD()
{
        CDatabase dataB;
        CString source=_T("SourceAccess"); // nom de la source
        dataB.Open(source);

        CRecordset rs(&dataB);
        CString query=_T("Select * from Categories where CatID<5");
        rs.Open(CRecordset::forwardOnly,query);

        while(!rs.IsEOF())
        {
                CString sValue1,sValue2;
                rs.GetFieldValue("Category",sValue1);
                rs.GetFieldValue("CatID",sValue2);
                //ajoute en tête du listBox
                m_liste.InsertString(0,sValue2+" "+ sValue1);
                rs.MoveNext();
        }
}
```

12.3 Définir la chaîne de connexion

La première solution testée consistait à accéder à une base de données en la référençant par son nom de source de données (DSN). Une autre solution consiste à fournir toutes les informations nécessaires à la connexion :

- nom du driver ODBC
- chemin et nom du fichier

Toutes ces informations sont contenues dans une chaîne de caractère ConnectStr où les champs sont séparés par des ;

ConnectStr="ODBC;Driver={ };DSN='';Dbq=d:\\Test.mdb;";

```
void CODBCDlg::OnLireBDD()
{
        // TODO: Add your control notification handler code here
        CDatabase dataB;
        CString ConnectStr;
        ConnectStr+=_T("ODBC;");
        ConnectStr+=_T("Driver={Microsoft Access Driver (*.mdb)};");
        ConnectStr+=_T("DSN='';");              //pas de source de donnée précisée
        ConnectStr+=_T("Dbq=d:\\Test.mdb;");    // fichier sous la racine du lecteur d:
        ConnectStr+=_T("Uid=;Pwd=;");

        dataB.Open(NULL,FALSE,FALSE,ConnectStr);

        CRecordset rs(&dataB);
        CString query=_T("Select * from Categories where Category<'G'");
        rs.Open(CRecordset::forwardOnly,query);

        while(!rs.IsEOF())
        {
```

```
            CString sValue1,sValue2;
            rs.GetFieldValue("Category",sValue1);
            rs.GetFieldValue("CatID",sValue2);
            m_liste.InsertString(0,sValue2+" "+ sValue1);
            rs.MoveNext();
      }
}
```

12.4 Créer une classe dérivée de `CRecordset` avec l'IDE Visual C++

On peut dériver `CRecordset` pour spécifier que la classe réalise une requête particulière sur une base donnée. Il suffit pour cela de créer, dans l'IDE Visual C++, une nouvelle classe ayant `CRecordset` comme classe de base, soit :

```
[Onglet Class View] | click droit sur ___ Classes | New Class
      Name = CMyRecord
      Base Class = CRecordset
```

Dès lors, un assistant vous permet de spécifier un nom de source de données (pour la base de données associée) et une table (pour générer une requête initiale).

La source de données, et la requête SQL, associées au record set sont précisés dans les méthodes virtuelles `GetDefaultConnect` et `GetDefaultSQL`. Grâce aux renseignements donnés dans les boîtes de dialogue de l'assistant, la classe spécialisée créée contient les implémentations suivantes :

```
CString CMyRecord::GetDefaultConnect()
{
      return _T("ODBC;DSN=SourceAccess"); // DSN de la base gérée
}

CString CMyRecord::GetDefaultSQL()
{
      return _T("[Categories]");       // Table sélectionnée
}
```

En outre la classe contient des attributs capables de stocker les données d'un enregistrement. Ici, la classe contient un attribut **`m_Category`** de type `CString` un autre `m_CatID` de type **`long`**.

On peut dès lors lire le résultat de la requête (qui est ici le contenu de la table `Categories`)

```
void CODBCWizDlg::OnLireBDD()
{
      /* la base est connue via
      la méthode GetDefaultConnect*/
      CMyRecord m_rec;

      /* la requete est désormais connue
      via la méthode GetDefaultSQL */
      m_rec.Open();

      // se placer en fin du recordset
      m_rec.MoveLast();

      while(!m_rec.IsBOF())
      {
      m_liste.InsertString(0,m_rec.m_Category);
      m_rec.MovePrev();
      }
}
```

Les implementations faites par défaut peuvent être modifiées à la main, si l'on connaît la syntaxe exacte de la chaîne de connexion. La sortie ci-contre correspond à l'implémentation ci-dessous

```
CString CMyRecord::GetDefaultConnect()
{
	// base de données Access (Test.mdb) dans le
	// répertoire courant
	return _T("ODBC;Driver={Microsoft Access Driver (*.mdb)};DSN='';Dbq=Test.mdb;");
}

CString CMyRecord::GetDefaultSQL()
{
	// requête SQL de sélection d'enregistrements
	return _T("Select * from Categories where Category>'G'");
}
```

12.5 L'ajout d'enregistrements

```
void CODBCWizDlg::OnAjout()
{
	// TODO: Add your control notification handler code here
	UpdateData();
	CMyRecord m_rec;
	m_rec.Open();

	//ajoute un enregistrement
	m_rec.AddNew();

	// édite l'enregistrement
	m_rec.m_CatID=m_CatID;
	m_rec.m_Category=m_Category;

	// met la base à jour
	m_rec.Update();
}
```

13 L'accès aux fichiers (sans la sérialisation)

On présente deux façons d'accéder à des fichiers dans des programmes Windows sous Visual C++ : l'accès grâce à un objet de la classe **fstream** (classe standard du C++, présente également dans la bibliothèque STL), l'accès grâce à un objet **CFile** (de la bibliothèque MFC).

13.1 Rappel : les déclarations d'énumérations au sein des classes

Avant de présenter l'utilisation des classes `fstream` et `CFile`, il est bon de rappeler comment les modes d'ouverture de fichiers sont représentés. Ceux-ci sont généralement désignés par des valeurs d'un type énumération défini au sein d'une classe (dans la partie à accès public).

La déclaration d'énumérations en C++ s'effectue par la syntaxe suivante (le type **bool** en est un exemple):

```
enum bool{false=0,true};
```

bool est un type, et **false/true** sont des valeurs pour le type **bool**. (valeurs numériques entières).

Une énumération définie au sein d'une classe représente une famille de constantes, référencées au moyen d'un symbole (ce qui est une alternative à la déclaration de constantes via la directive **#define**).

```
class Calendrier{
public:
 enum jour{lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche};
 Calendrier();
};
```

Le type énumération, ainsi que ses valeurs possibles, appartiennent à la classe `Calendrier`. C'est donc l'opérateur de résolution de portée :: qui permet de les référencer.

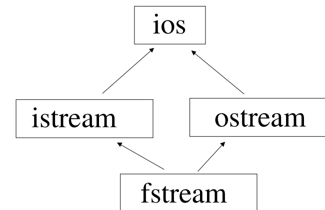
Calendrier::jour j=Calendrier::mardi;

`Calendrier::jour` est le type (le type `jour` est déclaré dans la classe `Calendrier`, d'où l'opérateur ::)
`Calendrier::mardi` est une des valeurs pour ce type énumération.

Cette technique est très couramment utilisée, notamment dans la bibliothèque STL pour les modes d'ouverture des flots et la définition des formats.

Remarque (la déclaration d'une classe au sein d'une classe) : ce qui vient d'être dit pour les énumérations est vrai pour n'importe quel type. Il est donc possible de déclarer une classe au sein d'une classe. C'est par exemple la technique utilisée pour les objets itérateurs des classes de la bibliothèque STL. Les classes `iterator` sont définies au sein des différentes classes `vector<T>, list<T>, deque<T>...`

13.2 La hiérarchie des classes de flots standards



Une vue (très partielle) de la hiérarchie des classes de gestion des flots entrée/sortie est décrite ci-contre.

La classe **ios** est la classe de base pour tous les flots. La classe **istream** est destinée aux flots d'entrée et **ostream** aux flots de sortie. D'ailleurs, **cin** et **cout** (utilisés dans les programmes console) sont des objets globaux instanciés à partir de ces classes. Enfin, la classe **fstream** permet la gestion des fichiers. On voit qu'elle dérive à la fois des deux classes **istream** et **ostream**.. Par conséquent, la gestion des fichiers via la classe **fstream** utilise des méthodes héritées des classes mères. C'est pourquoi on va survoler l'interface de ces classes.

ios::open_mode : énumération définie dans la classe **ios**. Associée en particulier aux modes d'ouverture des fichiers. Les valeurs possibles sont :

ios::in : ouverture en lecture
ios::out : ouverture en écriture
ios::app : ouverture en ajout (mode append)
ios::binary : ouverture en mode binaire. (Par défaut, on ouvre le fichier en mode texte.)

Ces modes peuvent être combinés par l'opérateur |, par exemple **ios::out|ios::app**.

Méthodes de la classe ostream :

ostream::put(char) : sortie non formatée d'un caractère
ostream::write(char *,int n) : sortie de n caractères
ostream::operator<<( ) : insertion formatée dans le flot pour certains types de base (char, int ...)

```
#include<fstream>
#include<iostream>

using namespace std;

void main()
{
	cout.put('a');
	cout.put('\n');
	char str[]={'a','b','c',0,'f','g','h'};
	cout << str << endl;
	cout.write(str,7);
}
```

Méthodes de la classe istream

`istream::get(char &)` : lecture d'un caractère
`istream::getline(char * pStr,int nbCarac,char delim='\n'`) : lecture d'une suite d'au plus **`nbCarac`** caractères. La lecture s'arrête aussi si le caractère délimiteur (par défaut '\n') est rencontré.
`istream::read(char * pStr,int nbCarac`) : lecture de **`nbCarac`** caractères.
`istream::operator>>(`) : lecture formatée pour les types de base.

```
using namespace std;
void main()
{
        char tampon[30];
        string str;

        cin.getline(tampon,30,'\n'); //la lecture s'arrête sur retour chariot
        cin >> str;         // la lecture s'arrête sur les espaces

        cout << tampon << endl;
        cout << str << endl;
}
```

13.3 La classe `fstream`

En C++, les fichiers peuvent être gérés par des flots de type **`fstream`** (file stream). La classe `fstream` hérite des méthodes des classes `istream` et `ostream`, donc de celles présentées précédemment. La classe `fstream` dispose des méthodes suivantes (ou héritées de ios, istream, ostream)

```
fstream::fstream() : crée un objet non connecté à un fichier

fstream::fstream(char * nomFichier, ios::open_mode ) : ouvre un fichier

fstream::~fstream() : fermeture d'un fichier ouvert

fstream::open(char *, ios::open_mode) : ouvre un fichier

fstream::close() : ferme le fichier en cours
```

Quelques équivalences entre les modes d'ouverture des flots et les modes d'ouverture de fichiers par fopen() en C

ios::open_mode	**mode pour la fonction fopen()**
ios::out	**w**
ios::in	**r**
ios::out|ios::app	**a**
ios::in|ios::out	**r+**
ios::in|ios::binary	**rb**
ios::out|ios::binary	**wb**
ios::out|ios::app|ios::binary	**ab**

```
void main()
{
        char tampon[150];

        // création ou remplacement du fichier
        fstream f("toto.txt",ios::out);

        f<<"test \t" << 1 << "\t"<< 1.5 << endl;    // écriture dans le fichier
        f.close();

        // réouverture du fichier en mode ajout
        f.open("toto.txt",ios::out|ios::app);
        f<<"ajout \t" << 2 << "\t"<< 2.5 << endl;   // écriture dans le fichier
        f.close();
```

```
        // ouverture en mode lecture
        f.open("toto.txt",ios::in);

        while(!f.eof())
        {
                f.getline(tampon,150);
                cout  << tampon << endl;
        }
}
```

L'accès direct dans les fichiers : la classe **`fstream`** hérite de méthodes de positionnement suivantes

Pour la lecture (hérité de istream):

```
fstream::seekg(déplacement, position) : positionne le pointeur
fstream::tellg() : retourne la position courante
```

Idem pour l'écriture (hérité de ostream) :

```
fstream::seekp(déplacement, position)
fstream::tellp()
```

La classe **`ios`** définit pour cela des valeurs **`ios::beg`** (début du flot), **`ios::end`** (fin du flot), **`ios::cur`** (position courante). Il est à noter que la lecture ou l'écriture font évoluer la position courante.

```
void main()
{
        char tampon[150];

// ouverture en écriture
        fstream f("fichier.txt",ios::out);
        for(int i=0;i<3;i++)  f<<"ligne numero "<< i << "\n";
// instant t1

f.seekp(0,ios::beg);                //positionne en début
        f<<"premiere ligne \n";
        f.close();

        // instant t2

// ouverture en lecture
        f.open("fichier.txt",ios::in);
//positionne au 3ième du début
        f.seekg(2*sizeof(char),ios::beg);
        f.getline(tampon,150);
        cout << tampon <<endl;

// déplace de 3 caractères
        f.seekg(3*sizeof(char),ios::cur);
        f.getline(tampon,150);
        cout << tampon <<endl;
}
```

```
buffer fichier.txt (instant t1)
--ligne numero 0--
--ligne numero 1--
--ligne numero 2--
```

```
buffer fichier.txt (instant t2)
1ere ligne
ro 0--
--ligne numero 1--
--ligne numero 2--
```

sortie

```
re ligne
0--
Press any key to continue
```

13.4 La classe `CFile`

La bibliothèque MFC dispose également d'une classe d'accès aux fichiers appelée `CFile`. L'utilisation suit les mêmes principes que la classe `fstream`, à savoir :

- les modes d'ouverture sont définis par une valeur d'un type énumération (`OpenFlags`), les valeurs possibles sont combinables par l'opérateur logique | , quelques valeurs possibles:

```
CFile::modeCreate   si le fichier existe, il est tronqué à 0
CFile::modeRead   (lecture seule)
CFile::modeReadWrite
CFile::modeWrite   (écriture seule)
```

- la classe dispose de méthodes de lecture/écriture et de positionnement (seek)

Exemple :

```
CFileException ex;   //exception
      CFile f;
      if(!f.Open("Fichier.txt",CFile::modeWrite,&ex))
      {
             TCHAR szError[1024];
             ex.GetErrorMessage(szError, 1024);
             ::AfxMessageBox(szError);
      }
      else
      {
             f.SeekToEnd();
             f.Write(m_strAjout,m_strAjout.GetLength());
             f.Close();
      }
```

Dans MFC, la méthode d'ouverture d'un fichier la plus courante est un processus à deux étapes.

Pour ouvrir un fichier

Créer l'objet fichier sans spécifier de chemin d'accès ou d'indicateurs d'autorisations.
Appeler la fonction membre **Open** pour l'objet fichier, en fournissant cette fois un chemin d'accès et des indicateurs d'autorisations. La valeur retournée pour **Open** est différente de zéro si l'ouverture du fichier a réussi ou égale à 0 s'il est impossible d'ouvrir le fichier spécifié. La fonction membre **Open** est conforme au prototype suivant :

```
virtual BOOL Open( LPCTSTR lpszFileName, UINT nOpenFlags, CFileException* pError = NULL );
```

Les indicateurs d'ouverture spécifient les autorisations que vous souhaitez associer au fichier, par exemple lecture seule. Les valeurs possibles des indicateurs sont définies en tant que constantes énumérées à l'intérieur de la classe `CFile` ; elles sont désignées par « **CFile::** », comme dans **`CFile::modeRead`**. Ainsi, l'indicateur **`CFile::modeCreate`** vous permet de créer le fichier.

Exemple : l'exemple suivant montre comment créer un fichier avec l'autorisation lecture/écriture (en remplaçant tout fichier existant ayant le même chemin d'accès) :

```
char* pszFileName = "c:\\test\\myfile.dat";
CFile myFile;
CFileException ex;

if ( !myFile.Open( pszFileName, CFile::modeCreate |
          CFile::modeReadWrite, &ex ) )
{
      TCHAR szError[1024];
      ex.GetErrorMessage(szError, 1024);
      ::AfxMessageBox(szError);
}
```

Pour lire et écrire dans le fichier, utiliser les fonctions membres `Read` et `Write`. La fonction membre `Seek` est également disponible pour atteindre un emplacement spécifique du fichier.

`Read` définit un pointeur vers un tampon et le nombre d'octets à lire et renvoie le nombre réel d'octets lus. Si le nombre d'octets nécessaire n'a pas pu être lu parce que la fin du fichier est atteinte, c'est le nombre réel d'octets lus qui est retourné. Si une erreur de lecture se produit, une exception est levée. Write est similaire à Read, si ce n'est

que le nombre d'octets écrits n'est pas retourné. Si une erreur de lecture se produit, notamment en cas d'impossibilité d'écrire tous les octets spécifiés, une exception est levée. Si vous avez un objet CFile valide, vous pouvez y lire et écrire conformément à l'exemple suivant :

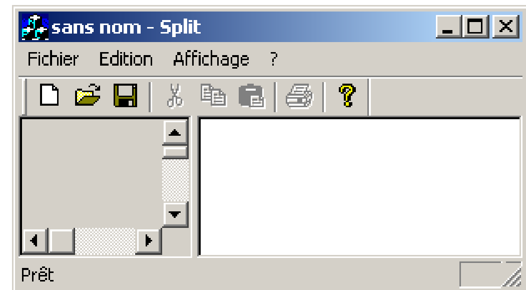
```
char      szBuffer[256];
UINT      nActual = 0;
CFile myFile;

myFile.Write( szBuffer, sizeof( szBuffer ) );
myFile.Seek( 0, CFile::begin );
nActual = myFile.Read( szBuffer, sizeof( szBuffer ) );
```

14 Exploitation avancée des vues

14.1 Fractionner la zone cliente (splitter)

Un splitter permet le découpage de la zone cliente en vues différentes. Par exemple, dans l'application ci-après, la zone cliente est séparée en deux vues, l'une dérivée de `CFormView` l'autre de `CView`.

Pour une application SDI, le splitter va être un objet membre de la classe de la fenêtre cadre CMainFrame (dérivée de CframeWnd).



Exercice pratique: réalisation de l'application ci-contre

1) Création d'un projet SDI (Split) avec CView comme classe de base

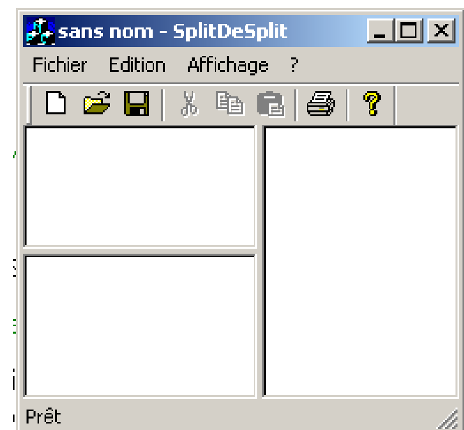
2) Création d'une ressource de forme et la classe CSplitFormView associée

A ce stade, `CSplitView` est la classe de vue (dérivée de `CView`) pour la zone cliente droite et `CSplitFormView` est la classe associée à la forme (zone cliente gauche).

3) Ajouter un objet `m_split` de type `CSplitterWnd` dans la classe `CMainFrame`

4) Ajouter , à l'aide ClassWizard, le gestionnaire suivant à la classe `CMainFrame`

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
// TODO: Add your specialized code here and/or call the base class
        BOOL rtn=m_splitter.CreateStatic(this,1,2); // découpage statique 1 ligne 2 colonnes
        rtn|=m_splitter.CreateView(0,0,RUNTIME_CLASS(CSplitFormView),CSize(200,0),pContext);
        rtn|=m_splitter.CreateView(0,1,RUNTIME_CLASS(CSplitView),CSize(0,0),pContext);
        return rtn;
}
```



La méthode `CSplitterWnd::CreateStatic()` crée la fenêtre splitter (fille de la fenêtre cadre) et son découpage (ici statique 1 ligne 2 colonnes). Ensuite, la méthode `CSplitterWnd::CreateView()` crée les différentes vues.

14.2 Fractionner une zone d'un splitter (splitters multiples)

Pour réaliser l'application ci-contre, il faut mettre deux objets de la classe `CSplitterWnd` dans la classe `CMainFrame`.

Le premier fait un découpage en 1 ligne deux colonnes. Le second

coupe la première colonne en deux lignes.

1) Ajouter deux objets `CSplitterWnd : m_main_split, m_sub_split à la classe CMainFrame`

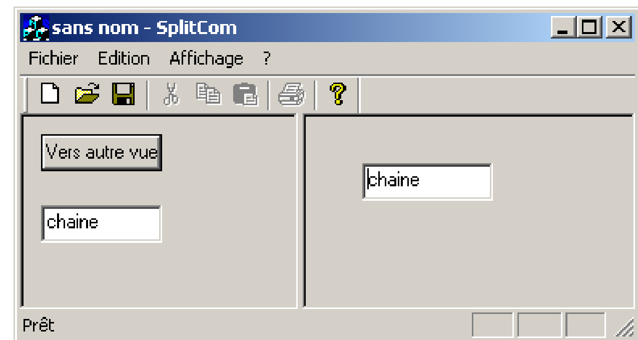
2) Dans la méthode `CMainFrame::OnCreateClient`, créer les splitters. Le premier splitter est fenêtre fille de la fenêtre cadre. Le second splitter est une fenêtre fille du premier splitter.

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
bool r=m_main_split.CreateStatic(this,1,2);
r|=m_main_split.CreateView(0,1,RUNTIME_CLASS(CSplitDeSplitView),CSize(200,0),pContext);

r|=m_sub_split.CreateStatic(&this->m_main_split,2,1,WS_VISIBLE|WS_CHILD|WS_BORDER);
r|=m_sub_split.CreateView(0,0,RUNTIME_CLASS(CSplitDeSplitView),CSize(100,100),pContext);
r|=m_sub_split.CreateView(1,0,RUNTIME_CLASS(CSplitDeSplitView),CSize(100,100),pContext);

m_main_split.SetActivePane(0,0);
return r;
}
```

14.3 Dialogue entre les vues dans le cas de fenêtres fractionnées

Lorsque le zone client est un splitter, les différentes vues ne peuvent pas dialoguer directement. En revanche, toutes les vues ont accès à l'objet document. Elles disposent toutes d'un pointeur sur l'objet document obtenu par la méthode `CView::GetDocument()`. C'est donc à travers le document que les fenêtres peuvent échanger des données. En outre, la classe de document possède une méthode `CDocument::UpdateAllViews()` pour raffraîchir les vues à partir des données du document.

Exercice pratique: la zone cliente de l'application ci-contre est divisée en deux vues dérivées de `CFormView`. La chaîne de la vue de gauche doit être recopiée dans la zone d'édition de la vue de droite lors d'un click sur le bouton.



1) créer l'application SDI avec splitter et deux vues de type `CFormView`.

   Ici les deux classes de vue sont **`CSplitComView`** et **`CFView`**.

2) Ajouter un objet `m_string_doc` (classe `CString`) dans la classe de document et associer un objet `CString` à chaque zone d'édition. Il y a donc un objet de type `CString` dans les deux classes de vue (`m_string_vue1` et `m_string_vue2`) et un dans la classe de document (`m_string_doc`).

3) Ajouter le gestionnaire d'événement `OnUpdate` à la classe de vue de droite

```
void CFView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
	m_string_vue2=GetDocument()->m_string_doc;
	UpdateData(FALSE);	// met à jour le contrôle
}
```

4) Ajouter le gestionnaire de click sur le bouton à la vue de gauche

```
void CSplitComView::OnButton()
```

```
{
	UpdateData(); // met à jour l'objet m_string_vue1
	GetDocument()->m_string_doc=m_string_vue1;
	GetDocument()->UpdateAllViews(this); // MAJ des vues
}
```

L'appel de `UpDateAllViews` sur le document va déclencher l'appel de la méthode `OnUpdate` de la vue2, ainsi que de toute vue qui aurait mis en place un tel gestionnaire.

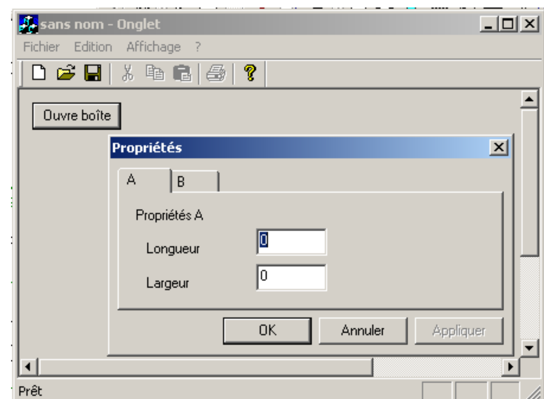
14.4 Boîte de dialogue à onglets (CPropertySheet)

La classe **`CPropertySheet`** gère les boîtes de dialogue à onglets. Chaque onglet est un objet d'une classe dérivée de **`CPropertyPage`**.

1) Créer une classe (avec Class Wizard) dérivée de **`CPropertySheet`**. Il s'agira de la classe de boîte de dialogue
2) Créer autant de ressources de type **`Dialog/ IDD_PROPPAGE`** que d'onglets
3) Associer une classe dérivée de **`CPropertyPage`** à chaque ressource (chaque onglet)
4) Ajouter, dans la classe dérivée de CPropertySheet, un objet membre par onglet
5) Dans le constructeur de la boîte de dialogue, ajouter les onglets par la méthode héritée

```
CPropertySheet ::AddPage(CPropertyPage *)
```

Exercice pratique: dans l'exemple ci-contre, la classe `COngletDlg` est dérivée de `CPropertySheet`. Chaque onglet est édité dans l'éditeur de ressource puis associé à une classe dérivée de `CPropertyPage`. Les deux classes CPP1 et CPP2 (dérivées de `CPropertyPage`) sont ainsi créées pour les onglets A et B.



Deux objets des classes `CPP1` et `CPP2` sont ajoutés à la classe COngletDlg.

```
class COngletDlg : public CPropertySheet
{
public:
	CPP2 m_page2; //Onglet A
	CPP1 m_page1; //Onglet B
};
```

Le constructeur de la classe `COngletDlg` ajoute les onglets à la boîte.

```
COngletDlg::COngletDlg(LPCTSTR  pszCaption,  CWnd*
pParentWnd, UINT iSelectPage)
	:CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
	this->AddPage(&m_page1);  // Ajoute Onglet A
	this->AddPage(&m_page2);  // Ajoute Onglet B
}
```

Ensuite, la boîte de dialogue est créée comme toute boîte de dialogue, excepté que le titre de la boîte est passé au constructeur.

```
void CForme::OnButton()
{
	COngletDlg dlg(_T("Propriétés"));
	dlg.DoModal();
}
```

Exploitation des champs saisis dans la boîte de dialogue à onglets

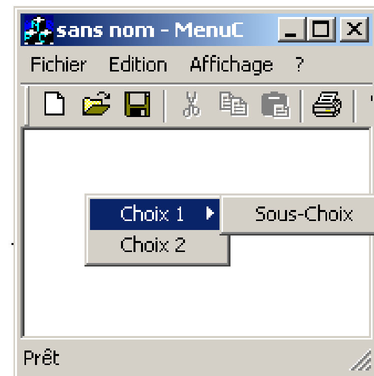
Il suffit d'associer (via Class Wizard) des variables aux contrôles des onglets. Ces variables apparaissent alors comme des membres publics des classes CPP1 et CPP2. Dès lors, l'accès à ces variables devient

```
COngletDlg dlg(_T("Propriétés"));
Dlg.m_page1.m_longueur ; // variable de la page 1 (onglet A) de la boîte dlg
```

L'exemple suivant affiche dans une boîte de message le contenu de tous les champs saisis dans les onglets :

```
void CForme::OnButton()
{
        COngletDlg dlg(_T("Propriétés"));

        if(dlg.DoModal()==IDOK)
        {
                // récupération des valeurs associées aux pages
                CString str1,str2;
                str1.Format("longueur =%d largeur =%d",
                        dlg.m_page1.m_longueur,
                        dlg.m_page1.m_largeur);
                str2.Format("x=%d y=%d",dlg.m_page2.m_x,dlg.m_page2.m_y);
                MessageBox(str1+str2);
        }
}
```

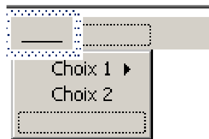
14.5 Menu contextuel



Les menus contextuels sont les menus qui apparaissent généralement à l'emplacement du curseur lors d'un click droit.

Pour mettre en place un tel menu, il faut

1) éditer une nouvelle ressource **IDR_MENUCONTEXT** (l'ID est libre) de menu dans l'éditeur de ressources. Mettre les commandes dans le premier sous-menu (peu importe le titre du menu)



2) Mettre en place un gestionnaire de message `WM_CONTEXTMENU`

3) Compléter le gestionnaire de message de la façon suivante

```
void CMenuCView::OnContextMenu(CWnd* pWnd, CPoint point)
{
CMenu menu;
menu.LoadMenu(IDR_MENUCONTEXT);
menu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN|TPM_RIGHTBUTTON,
                                    point.x,point.y,this);
}
```

4) Il suffit alors de mettre en place des gestionnaires de message associés aux commandes du menu, par exemple

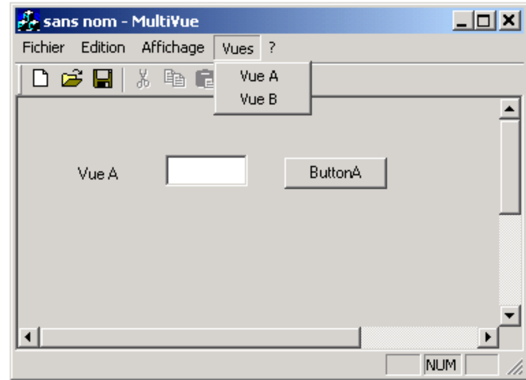
```
BEGIN_MESSAGE_MAP(CMenuCView, CView)
ON_COMMAND(ID__CHOIX1, OnChoix1)
END_MESSAGE_MAP()

void CMenuCView::OnChoix1()
{
        MessageBox("hello");
}
```

14.6 Changement dynamique de vue dans la zone cliente

Le changement dynamique de vue dans la zone cliente nécessite la création de plusieurs classes de vue différentes (une classe par vue). Une commande du menu permettra ensuite de basculer d'une vue à une autre.

Lorsque l'on veut que les vues soient des formes, il convient de créer une ressource (`insert/dialog/IDD_FORMVIEW`) par forme puis de générer la classe associée (choisir `CFormView` comme classe de base) à l'aide de `ClassWizard`.

Note : il convient de rendre le constructeur de la classe dérivée de CFormView public (il est par défaut déclaré protégé).



Un objet persistant (alloué dans le tas) doit être instancié pour chaque vue. Ces objets existeront tout au long de l'exécution de l'application.

En pratique, lorsque l'on choisit l'architecture Doc/View, un objet vue est automatiquement créé (en relation avec le document et le fenêtre cadre) par le DocTemplate (durant l'exécution de C___App::InitInstance()). Aussi, pour avoir deux vues différentes (échangeables dynamiquement) il faut créer une seconde vue, c'est-à-dire un second objet persistant d'une classe de vue, et le mettre également en relation avec le document existant.

Pour cela, il faut créer un objet de la classe CreateContext, indiquer à cet objet quel est le document en cours (déja créé), puis créer la nouvelle vue en passant l'objet CreateContext en argument. Ceci permet à la nouvelle vue d'être également en relation avec le document.

On va donc ajouter deux variables membres **`C___App::m_pVue1`** et `C___App::m_pVue2` de type `CView *` à la classe d'application. A la fin de l'exécution de `C__App::InitInstance()`, chaque pointeur contiendra l'adresse d'un objet vue. Ensuite, changer de vue consistera à modifier la visibilité des vues et à modifier la fenêtre active.

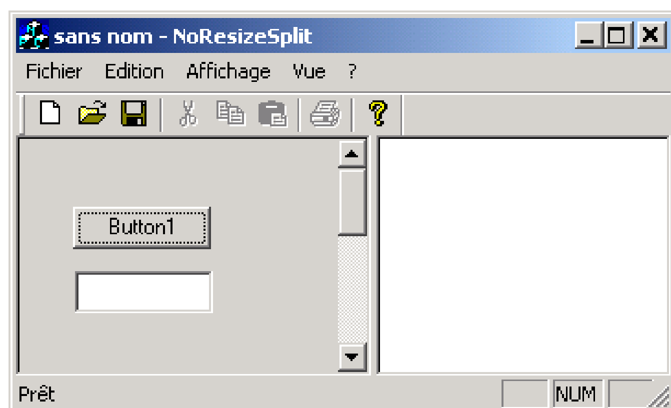
```
BOOL CMultiVueApp::InitInstance()
{
	AfxEnableControlContainer();
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));
	LoadStdProfileSettings();  // Load standard INI file options (including MRU)

	CSingleDocTemplate* pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
		IDR_MAINFRAME,
		RUNTIME_CLASS(CMultiVueDoc),
		RUNTIME_CLASS(CMainFrame),       // main SDI frame window
		RUNTIME_CLASS(CMultiVueView));
	AddDocTemplate(pDocTemplate);

	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;

	// A ce stade, le document et une vue sont créés
	// m_pVue1 = adresse de l'objet vue créé par le DocTemplate
	CView* pActiveView = ((CFrameWnd*) m_pMainWnd)->GetActiveView();
	m_pVue1 = pActiveView;

	// m_pVue2 = adresse d'un objet vue alloué dans le tas
	m_pVue2 = (CView*) new CSecondeVue;

	CDocument* pDoc = ((CFrameWnd*)m_pMainWnd)->GetActiveDocument();
	CCreateContext context;
```

```
        context.m_pCurrentDoc = pDoc;

        // L'ID de la vue active initiale est AFX_IDW_PANE_FIRST.
        // Icrémenter cet ID de 1 fonctionne pour les architectures Doc/View mais
        // pas pour les splitters
        UINT viewID = AFX_IDW_PANE_FIRST + 1;
        CRect rect(0, 0, 0, 0);          // sera redimensionné plus tard.

        // Create the new view. L'objet est persistant mais sera détruit par l'application
        m_pNewView->Create(NULL,NULL, WS_CHILD|WS_BORDER,rect,m_pMainWnd,viewID, &context);

        // Quand le doctemplate crée la vue, le message WM_INITIALUPDATE est envoyé à
        //la vue. Il faut le faire explicitement ici.
        m_pNewView->SendMessage(WM_INITIALUPDATE, 0, 0);

        // The one and only window has been initialized, so show and update it.
        m_pMainWnd->ShowWindow(SW_SHOWMAXIMIZED);
        // The one and only window has been initialized, so show and update it.
        m_pMainWnd->ShowWindow(SW_SHOW);
        m_pMainWnd->UpdateWindow();

        return TRUE;
}
```

Le changement dynamique de vue consiste alors à basculer d'une vue à une autre. Pour celà, on a mis en place des gestionnaires sur les commandes du menu.

```
void CMultiVueApp::OnVueA()
{
        UINT ID1=m_pVue1->GetDlgCtrlID();       //échange des ID des deux vues
        UINT ID2=m_pVue2->GetDlgCtrlID();       // pour que RecalcLayout() fonctionne
        m_pVue1->SetDlgCtrlID(ID2);
        m_pVue2->SetDlgCtrlID(ID1);

        m_pVue2->ShowWindow(SW_HIDE);    // changer la visibilité
        m_pVue1->ShowWindow(SW_SHOW);
        ((CFrameWnd*)m_pMainWnd)->SetActiveView(m_pVue1);
        ((CFrameWnd*)m_pMainWnd)->RecalcLayout();
        m_pVue1->Invalidate();
}
void CMultiVueApp::OnVueB()
{
        // TODO: Add your command handler code here
        UINT ID1=m_pVue1->GetDlgCtrlID();
        UINT ID2=m_pVue2->GetDlgCtrlID();
        m_pVue1->SetDlgCtrlID(ID2);
        m_pVue2->SetDlgCtrlID(ID1);

        m_pVue1->ShowWindow(SW_HIDE);
        m_pVue2->ShowWindow(SW_SHOW);
        ((CFrameWnd*)m_pMainWnd)->SetActiveView(m_pVue2);
        ((CFrameWnd*)m_pMainWnd)->RecalcLayout();
     m_pVue2->Invalidate();
}
```

A ce stade chaque vue est associée au même document, le reste du développement pour l'exploitation de contrôles reste identique.



14.7 Empêcher le changement de position d'un splitter

Pour que le splitter ne puisse pas être déplacé par l'utilisateur, il faut dériver la classe `CSplitterWnd` et intercepter le message `WM_NCHITTEST` dans la classe dérivée.

Lors de chaque mouvement de la souris, Windows envoie un message WM_NCHITTEST à la fenêtre pour lui demander où se trouve la souris (dans l'aire client, sur la barre de titre, sur la bordure, etc.). En fonction de la réponse, Windows peut entreprendre une action (par exemple déplacer la fenêtre lorsque l'on clique sur la barre de titre et déplace la souris, bouton enfoncé). Généralement, un programme ne traite pas ce message. Nous allons le faire. Nous tromperons Windows en lui faisant croire que le curseur ne se trouve pas sur le splitter. Conséquence : Windows ne sachant pas que la souris se trouve sur le splitter, il ne pourra pas le déplacer!

Il suffit de dériver la classe CSplitterWnd et d'implémenter un comportement particulier pour le gestionnaire de message WM_NCHITTEST.

```
// extrait du fichier .h
class CMySplitter : public CSplitterWnd
{
DECLARE_DYNAMIC(CMySplitter)
DECLARE_MESSAGE_MAP()
public:
afx_msg UINT OnNcHitTest(CPoint point);
CMySplitter();
virtual ~CMySplitter();
};
```

```
// extrait du fichier .cpp
IMPLEMENT_DYNAMIC(CMySplitter,CSplitterWnd)

BEGIN_MESSAGE_MAP(CMySplitter,CSplitterWnd)
	ON_WM_NCHITTEST()
END_MESSAGE_MAP()

CMySplitter::CMySplitter()
{}

CMySplitter::~CMySplitter()
{}

UINT CMySplitter::OnNcHitTest(CPoint point)
{
	return HTNOWHERE;
}
```

Ensuite, il suffit d'utiliser cette classe pour gérer l'objet splitter membre de la classe de la fenêtre cadre.

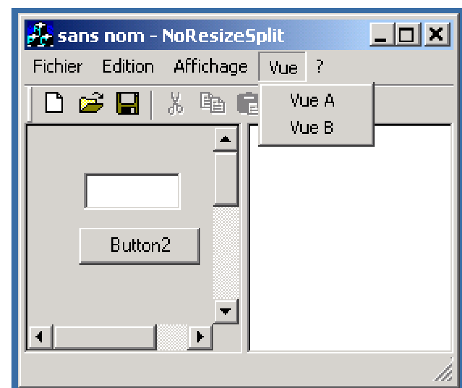
```
class CMainFrame : public CFrameWnd
{
	//...
public:
	CMySplitter m_splitter;
};
```

La création des vues du splitter reste identique en utilisant les méthodes héritées de CSpliterWnd (revoir la partie fractionner la fenêtre cliente)

14.8 Changement dynamique de vue dans un splitter

Il est également possible de modifier les vues dans les splitters. Une façon de procéder est de détruire et ré-instancier un objet persistant lors de chaque changement de vue. La différence principale avec le changement de vue dynamique dans la zone cliente est que les objets vue n'existent plus lorsqu'ils ne sont pas visibles.

En revanche, chaque vue sera également associée au document en cours.




```
void CMainFrame::OnVueA()
{
	m_splitter.DeleteView(0,0);
	CRect cr;
	GetClientRect(&cr);
	CDocument * pDoc=this->GetActiveDocument();
	CCreateContext cc;
	cc.m_pCurrentDoc=pDoc;
```

```
	m_splitter.CreateView(0,0,RUNTIME_CLASS(CFormeA),
				CSize(cr.Width()/4,cr.Height()),
				&cc);
	m_splitter.RecalcLayout();
}

void CMainFrame::OnVueB()
{
	m_splitter.DeleteView(0,0);
	CRect cr;
	GetClientRect(&cr);
	CDocument * pDoc=this->GetActiveDocument();
	CCreateContext cc;
	cc.m_pCurrentDoc=pDoc;
	m_splitter.CreateView(0,0,RUNTIME_CLASS(CFormeB),
				CSize(cr.Width()/4,cr.Height()),
				&cc);
	m_splitter.RecalcLayout();
}
```

Table des matières

1 Introduction 2
2 Rappels de C++ 2
2.1 Fiche 1 : Les classes en C++ 3
2.2 Fiche 2 : Gestion des projets avec des classes en C++ 4
2.3 Fiche 3 : Introduction aux classes standard (STL) et aux classes paramétrées en type 5
2.4 Fiche 4 : Réalisation de la relation de composition en C++ 6
2.5 Fiche 5 : Réalisation de la relation de spécialisation en C++ 7
2.6 Fiche 6 : Le polymorphisme en C++ 7
2.7 Fiche 7 : Les membres statiques 9
2.8 Fiche 8 : Les patrons de classes en C++ 10
2.9 Le RTTI (Run Time Type Information) et les nouvelles syntaxes de cast 11
2.10 Le préprocesseur 12
3 Environnement de développement Visual C++ 13
4 Notion de programmation événementielle sous Windows - API Windows 14
5 Introduction aux Microsoft Foundation Classes (MFC) (sans AppWizard ni ClassWizard) 17
5.1 Premier exemple utilisant les MFC 17
5.2 Exemple d'application dont la fenêtre principale est une boîte de dialogue 19
5.3 Exemple avec une fenêtre cadre et une fenêtre fille contenant un texte 20
5.4 Exemple de classe de fenêtre cadre avec gestionnaires de messages 20
5.5 Les macros de gestion de la table des messages 22
5.6 Qu'avons-nous appris jusqu'ici ? 23
6 Programmation MFC en utilisant les assistants de Visual C++ 24
6.1 Application « Dialog Based » réalisée avec l'assistant AppWizard 24
6.2 Modifier la ressource de boîte de dialogue principale 25
6.3 Ajouter des gestionnaires de message à l'aide de ClassWizard 26
6.4 Supprimer un gestionnaire de message à l'aide de ClassWizard 27
6.5 Exploiter les contrôles dans une boîte de dialogue 28
6.5.1 Associer un objet de type `CWnd` à un contrôle avec ClassWizard 28
6.5.2 Exercice de synthèse : création du projet, modification des ressources, ajout de gestionnaires de messages, exploitation des contrôles 29
6.5.3 Associer une variable à un contrôle avec ClassWizard 30
6.5.4 Retrouver l'objet `CWnd` associé à un contrôle grâce à son identifiant (ID) 32
6.6 Association d'un même gestionnaire de messages à plusieurs contrôles 33
6.7 Ajout d'un menu à une boîte de dialogue 34
6.8 Les Timers 34
6.9 Ajout d'une boîte de dialogue 35
7 Les boîtes de dialogue 37
7.1 Les boîtes de messages 37
7.2 Les boîtes « Modales » 38
7.2.1 Interception des clicks sur les boutons OK/CANCEL de la boîte 38
7.3 Les boîtes de dialogue « Non Modales » (Modeless) 39
7.3.1 Ajouter un membre de type `CNonModaleDlg` à la classe de la boîte de dialogue principale 39
7.3.2 Créer un objet dynamique de la classe `CNonModaleDlg` 42
7.3.3 Détecter les fuites mémoire avec Visual C++ 43
7.3.4 Créer des objets alloués dynamiquement de la classe `CNonModaleDlg` et leur assurer une « sorte » de désallocation « automatique » 43
7.3.5 Comment empêcher la création d'objets automatiques d'une classe 44
8 Application ayant une architecture Modèle-Vue-Contrôleur (MVC) 46
8.1 Introduction 46
8.2 Les modèles de conception (design patterns) 47
8.3 La partie Modèle 47
8.4 Modifier la classe `Repertoire` suivant le modèle de conception Singleton 49
8.5 Rappel sur les classes abstraites (notion d'interface) 51
8.6 Interface de classe `Repertoire` 52
8.7 Modifier la classe `Fiche` pour que seuls des objets alloués dans le tas puissent être créés 55

8.8 Réaliser une application de type boîte de dialogue utilisant le singleton `Repertoire` 56
8.9 Le modèle de conception Sujet-Observateurs 56
8.10 Application finale 59
9 La sérialisation 61
10 Les applications MFC utilisant le modèle Document-Vue 66
10.1 Introduction 66
10.2 Procédure de génération d'une application SDI avec AppWizard 66
10.3 Rôle des classes du modèle SDI 67
10.4 Séquence de création des objets d'une application SDI 70
10.5 Changer la classe de vue d'une application SDI 72
10.6 Exploiter une classe métier dans une application SDI avec l'architecture Document/View 72
10.7 Les applications MDI (Muli Document Interface) 76
11 Programmation réseau (les sockets) 78
11.1 Le modèle client-serveur, interface sockets 78
11.2 La classe CAsyncSocket 78
11.3 Chronologie des opérations Client-Serveur 80
11.4 Utilisation de l'API Socket avec les MFC 82
11.5 Exemple d'application Boîte de Dialogue avec utilisation d'un socket client 82
12 Accès aux bases de données avec ODBC 83
12.1 ODBC en deux mots 83
12.2 Les classes CDatabase et CRecordset 84
12.3 Définir la chaîne de connexion 85
12.4 Créer une classe dérivée de `CRecordset` avec l'IDE Visual C++ 86
12.5 L'ajout d'enregistrements 87
13 L'accès aux fichiers (sans la sérialisation) 87
13.1 Rappel : les déclarations d'énumérations au sein des classes 87
13.2 La hiérarchie des classes de flots standards 88
13.3 La classe `fstream` 89
13.4 La classe `CFile` 90
14 Exploitation avancée des vues 92
14.1 Fractionner la zone cliente (splitter) 92
14.2 Fractionner une zone d'un splitter (splitters multiples) 92
14.3 Dialogue entre les vues dans le cas de fenêtres fractionnées 93
14.4 Boîte de dialogue à onglets (CPropertySheet) 94
14.5 Menu contextuel 95
14.6 Changement dynamique de vue dans la zone cliente 96
14.7 Empêcher le changement de position d'un splitter 97
14.8 Changement dynamique de vue dans un splitter 98

MCours.com