

# CorTeX 2K5

# SOMMAIRE

## **1<sup>er</sup> Partie – Description des fonctionnalités des PDA en général**

### **Introduction**

#### **1. Fonctionnalités classiques des PDA**

## **2<sup>ème</sup> Partie – Cahier des charges**

### **1. Sujet**

### **2. Fonctionnalités implémentées**

## **3<sup>ème</sup> Partie – Analyse**

### **1. Descriptifs des fonctionnalités**

### **2. Hiérarchie de l'application**

## **4<sup>ème</sup> Partie – Programmation**

### **1. Les objets de base**

### **2. Les structures de données**

### **3. Le système de sauvegarde**

### **4. Problèmes rencontrés**

## **Conclusion**

# 1<sup>ère</sup> Partie – Description des fonctionnalités des PDA en général

## Introduction

L'engouement pour les assistants numériques personnels (PDA) ne cesse de prendre de l'ampleur depuis plusieurs années maintenant. Conçus exclusivement pour les professionnels à leurs débuts ces derniers se démocratisent désormais dans les foyers où ils sont de plus en plus prisés. Leur objectif premier est de permettre à leurs utilisateurs une meilleure organisation de leur travail. Qu'il s'agisse d'informations sur les contacts ou de la gestion des rendez-vous, un bon Pda doit être simple, rapide et sûr.

Avec les années, les progrès techniques ont conduit les assistants numériques à devenir de véritables plates-formes multimédias dotées d'interfaces graphiques toujours plus belles, et de fonctionnalités toujours plus avancées tel que nous le verrons dans le paragraphe suivant.

## 1. Fonctionnalités classiques des PDA

Comme demandé dans le sujet du projet, nous nous proposons dans cette partie, de faire l'état des lieux de l'offre actuelle des fonctionnalités des assistants personnels. Il existe de nos jours beaucoup d'assistants numériques, essayant de proposer toujours plus de fonctionnalités afin de se distinguer de leur concurrent. De nos jours, les ordinateurs de poche possèdent de véritables systèmes d'exploitation, et dans ce domaine les deux grands ténors du marché sont Apple (constructeur du premier Pda en 1993, le Newton) avec *Palm OS* et Microsoft avec *Windows Mobile*. Ces deux constructeurs ont une approche assez différente du monde des assistants numériques mais ils n'en proposent pas moins des fonctionnalités communes que nous listons ci-après :

- La gestion d'un « répertoire » de contacts, dans lequel est stockée une liste de contacts, et diverses informations les concernant.
- La gestion d'un « organisateur » appelé parfois « agenda », dans lequel est stockée une liste de rendez-vous.
- Un ensemble d'utilitaires pratiques, tels qu'une calculatrice, un aide mémoire et des jeux. Cependant, avec les années, la panoplie d'utilitaires ne cesse de s'étoffer. En effet, il est désormais commun d'y trouver un lecteur de musiques et de vidéos, une boîte de réception d'e-mails, des jeux vidéos, mais également des tableurs et des traitements de textes (à l'instar d'Excel et de Word, présents dans les Pda avec Windows Mobile).
- Un système de sécurité : les informations contenues dans les PDA sont généralement très sensibles. Ainsi, afin de garantir une sécurité minimale, les

PDA proposent un système de login / mot de passe, et diverses informations sur le propriétaire.

- Un menu de configuration des informations système, permettant à l'utilisateur de personnaliser l'interface de son assistant numérique et redéfinir son login / mot de passe et ses informations personnelles.

## 2<sup>ème</sup> Partie – Cahier des charges

### 1. Sujet

Le projet consiste à étudier les fonctionnalités proposées par les assistants numériques personnels disponibles dans le commerce, à sélectionner et implémenter une partie de ces fonctionnalités.

Le travail sera décomposé en 3 parties :

- étude des fonctionnalités classiques des PDA disponibles dans le commerce,
- sélection d'un ensemble cohérent de fonctionnalités parmi celles étudiées (avec au minimum la gestion de l'agenda de l'utilisateur et une gestion de la liste des contacts de l'utilisateur,
- Implémentation du PDA.

Le projet sera obligatoirement implémenté en Visual C++ (sans interface graphique). Les différentes options seront disponibles via des options de menus (sous forme textuelle).

Les données seront stockées dans des fichiers. On pourra soit travailler directement sur les fichiers et/ou soit charger les fichiers dans des structures en mémoire au début du programme, et les sauvegarder à la fin, selon les traitements à réaliser sur ces données.

L'implémentation proposée devra mettre en œuvre le mécanisme d'exceptions proposé par C++ pour gérer certains cas d'erreur (à préciser).

Les contacts de l'utilisateur seront gérés dans le programme à l'aide d'une liste *générique*. Les données (informations sur les contacts) seront lues et chargées au début de l'application et sauvegardées à la fin.

### 2. Fonctionnalités implémentées

Pour le choix des fonctionnalités à implémenter, nous avons sélectionné les plus pratiques, les plus utiles, et les plus caractéristiques d'un ordinateur de poche en leur apportant des méthodes avancées pour assurer une meilleure efficacité tout en conservant une utilisation intuitive pour l'utilisateur. Ainsi, les fonctionnalités retenues pour CorTeX2K5 sont :

- Un Agenda permettant la gestion de rendez-vous
- Un Répertoire, permettant la gestion d'une liste de contacts
- Quelques Utilitaires pratiques tels que :
  - une Calculatrice
  - un Block note

- un Aide Mémoire
- Un menu Système, permettant de définir les informations sur le propriétaire et sur l'interface console du programme
- Une protection par mot de passe au lancement du Pda
- Un Système d'aide complet
- L'affichage constant de l'arborescence des menus, permettant une navigation simplifiée pour l'utilisateur

Nous verrons plus en détails les méthodes relatives à chaque fonctionnalité dans la partie suivante.

## 3<sup>ème</sup> Partie – Analyse

Nous avons créé le Pda de façon très modulaire, en respectant le plus possible les avantages de la Programmation Orientée Objet offerte par C++. Ainsi, il serait possible d'apporter de nouvelles fonctionnalités à notre application seulement en les ajoutant à notre classe principale.

### 1. Descriptif des fonctionnalités

#### 1.1 La Classe principale

Afin de réunir toutes les fonctionnalités du Pda nous avons défini une classe principale (*CPda*). Cette classe ne possède qu'une fonction *main()* dans laquelle est présente une instance de chaque fonctionnalité implémentée. C'est également cette classe qui contrôle le système de mot de passe, ainsi que l'affichage des rendez-vous du jour et/ou anniversaire. Cette classe possède donc une instance de chaque fonctionnalité offerte par l'application, à savoir :

- Le répertoire (CRepertoire)
- L'agenda (CAgenda)
- La classe d'Utilitaires (CUtilitaires)
- Le Système (CSysteme)

Dès que l'utilisateur a choisi le menu dans lequel il veut aller, la classe *CPda* appelle la méthode *menu()* de la fonctionnalités désirée.

#### 1.2 Gestion du répertoire

Le répertoire nécessite trois classes. La première (*CContact*) est une classe de définition d'un contact, la seconde (*CListeContact*) est une classe héritière de la liste générique (*CListe*) et apporte de nouvelles méthodes typiques de la gestion de contacts telles que la recherche selon plusieurs critères, et des statistiques sur les contacts.

Enfin la dernière (*CRepertoire*) est une classe gérant les différents menus du répertoire tels que :

- L'ajout d'un nouveau contact
- L'affichage de tous les contacts
- La modification d'un contact
- La recherche d'un contact (selon plusieurs critères, voir la partie 2.2.1 pour plus de détails)
- La suppression d'un contact
- L'affichage des statistiques sur la liste de contacts

### 1.3 Gestion de l'Agenda

L'agenda nécessite cinq classes. Les trois premières classes ont la même structure que celles qui sont présente pour le répertoire. Respectivement nous avons les classes *CRendezVous*, *CListeRendezVous* qui hérite également de la classe *CListe* et *CAgenda*. Afin d'augmenté la modularité du projet nous avons géré les rendez-vous avec l'aide des classes *CDate* et *CHeure* qui permettent la gestion des dates et des heures en vérifiant leur validités (année bissextiles, etc.) Voyons le détails des menus de la classe *CAgenda* :

- Ajout d'un rendez-vous
- Recherche d'un rendez-vous (par Date ou parType)
- Suppression d'un rendez-vous
- Modification d'un rendez-vous
- Affichage des rendez-vous (du jour, d'une date donnée, ou de tous)

### 1.4 Les Utilitaires

Pour regrouper les utilitaires, nous avons eu recours à une classe *CUtilitaire*. Celle-ci possède une instance de chaque utilitaires présents dans le Pda, à savoir :

- Une Calculatrice, qui permet d'effectuer des additions, des multiplications, des soustractions et des divisions.
- Une Aide mémoire, offrant un système de gestion de Pense Bête définie par :
  - un Titre
  - un Type (course, travail, idée...) (facultatif).
  - une Description (facultative).
- Un Block note, simulant la gestion de fichier texte.

### 1.5 Configuration du système

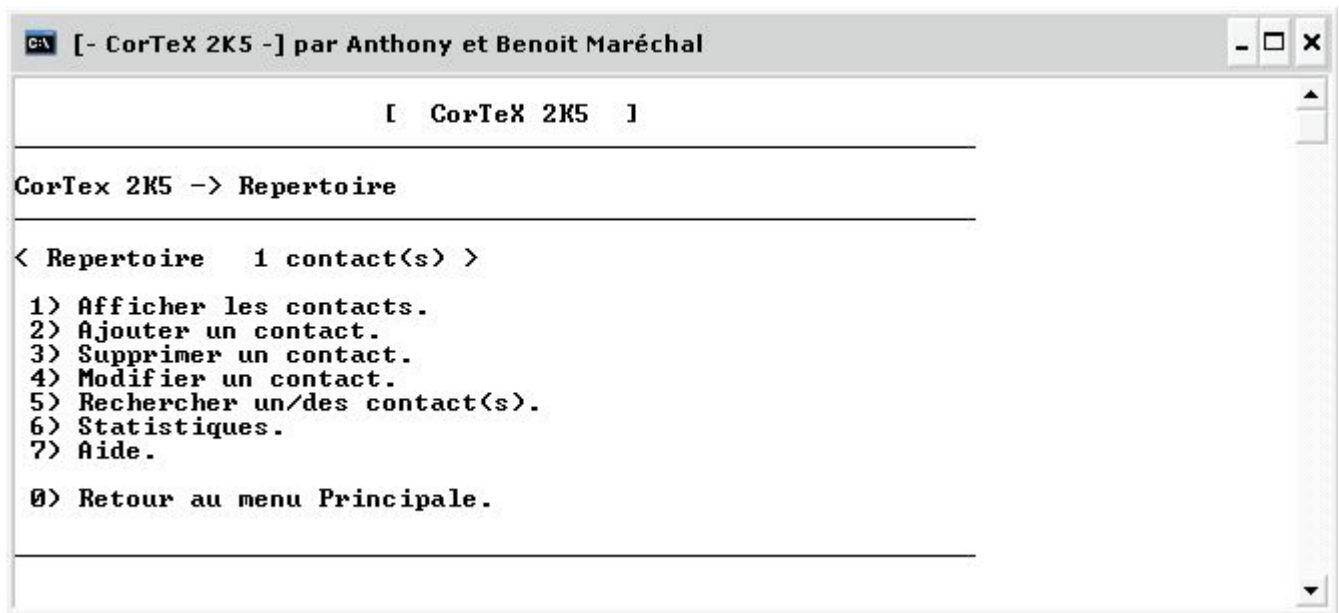
Comme tout bon Pda, Cortex2K5 permet la configuration des informations systèmes grâce à la classe (*CSysteme*). Elle consiste ici en diverses informations sur le propriétaire du Pda, définie dans la classe *CProprietaire* ; elle permet également de définir l'apparence graphique du Pda. Notre projet étant en interface console, ces dernières sont donc limitées au choix de la couleur de fond et de la couleur du texte.

## 1.6 Système d'aide et navigation

Pour rendre notre assistant personnel le plus simple et le plus clair possible, nous lui avons apporté un système d'aide performant, et très accessible, ainsi qu'une navigation simplifier.

Le menu principal et tous ses sous-menus disposent d'une fonction d'aide, ce qui est également le cas des sous-fonctionnalités dont le fonctionnement ne serait pas intuitif pour l'utilisateur (interface console oblige...).

Enfin, pour que l'utilisateur ne soit jamais perdu dans les divers sous-menus de l'application, nous affichons constamment dans l'en-tête, l'arborescence des menus (voir screenshot ci-dessous).



```
C:\ [- CorTeX 2K5 -] par Anthony et Benoit Maréchal

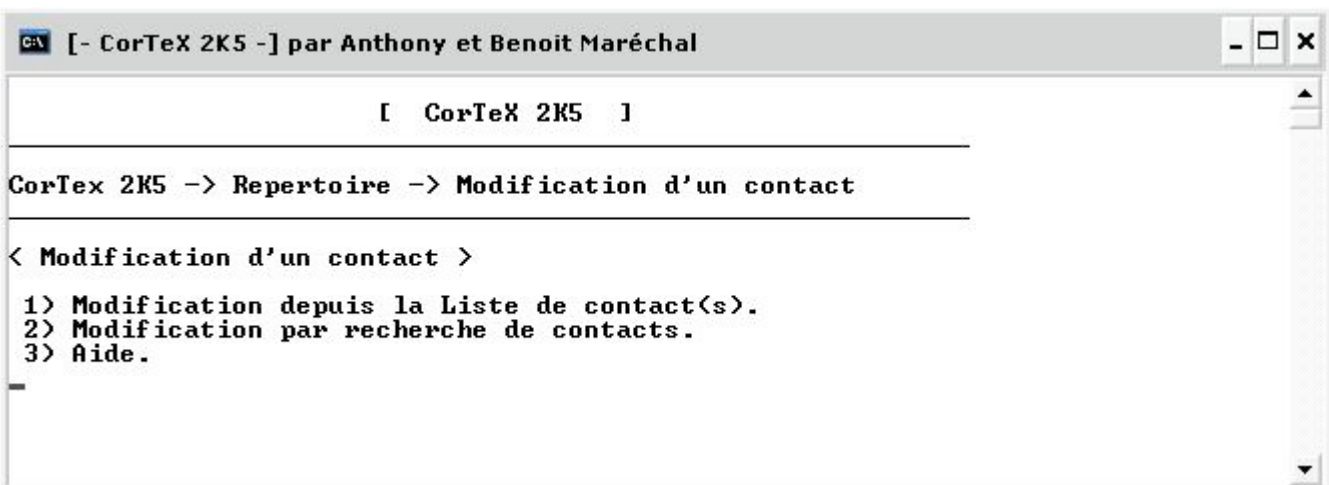
[ CorTeX 2K5 ]

CorTex 2K5 -> Repertoire

< Repertoire 1 contact(s) >

1) Afficher les contacts.
2) Ajouter un contact.
3) Supprimer un contact.
4) Modifier un contact.
5) Rechercher un/des contact(s).
6) Statistiques.
7) Aide.

Ø) Retour au menu Principale.
```



```
C:\ [- CorTeX 2K5 -] par Anthony et Benoit Maréchal

[ CorTeX 2K5 ]

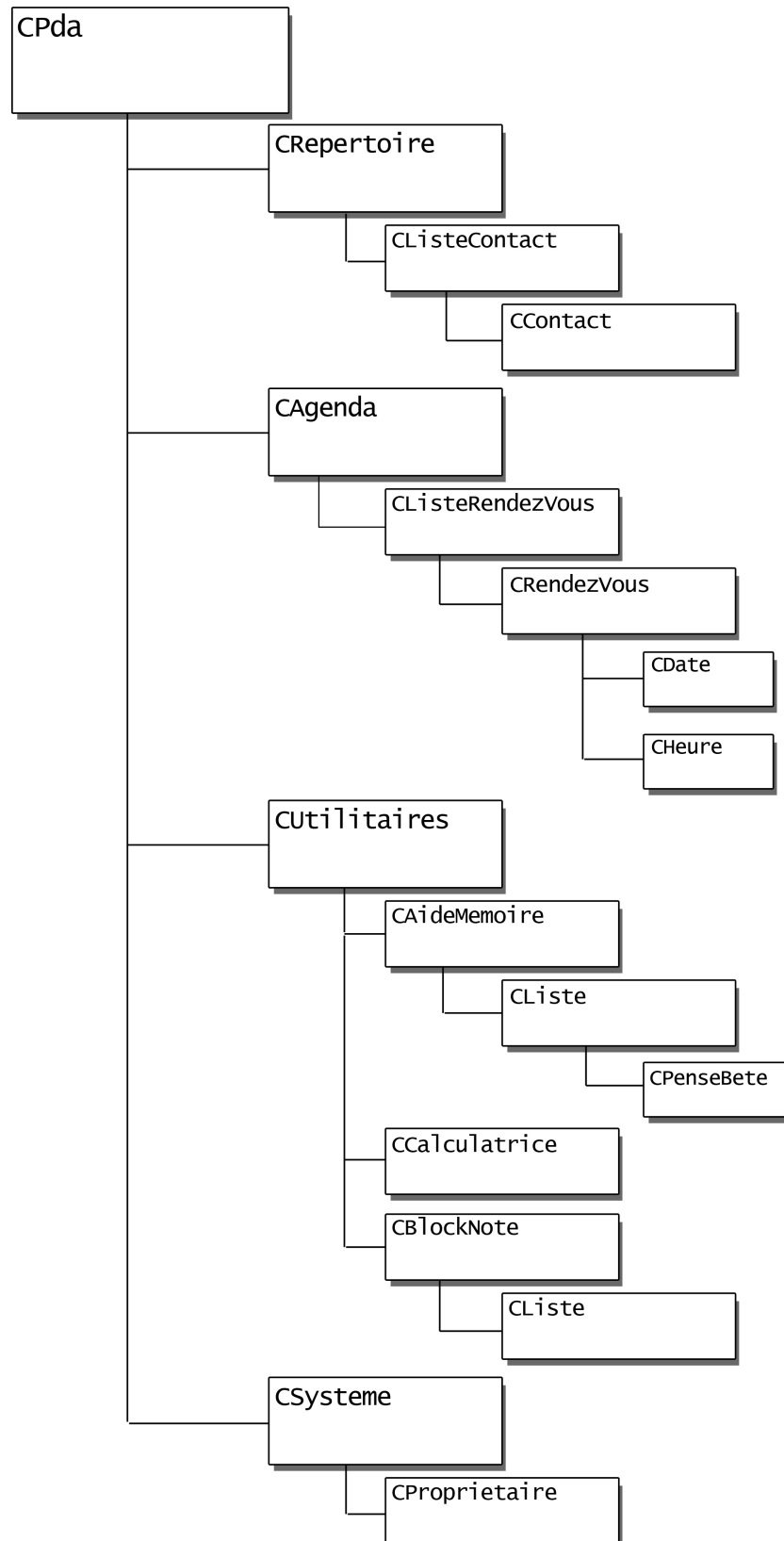
CorTex 2K5 -> Repertoire -> Modification d'un contact

< Modification d'un contact >

1) Modification depuis la Liste de contact(s).
2) Modification par recherche de contacts.
3) Aide.
```

## 2. Hiérarchie de l'application

Afin d'avoir une meilleure vision de notre programme, nous avons réalisé le schéma de la hiérarchie des différentes classes :





## 4<sup>ème</sup> Partie – Programmation

### 1. Les objets de bases

Nous allons voir dans cette partie le code source du fichier d'en-tête de la classe *CContact* et la description des en-têtes des autres objets de base, tels que les classes définissant un Rendez-vous, un Pense-Bête et le Propriétaire. Ces classes étant très similaires entre elles, nous donnerons seulement le code source de la classe *CContact* (fichier .h et .cpp).

#### 1.1 Les entêtes des objets de base

##### 1.1.1 La classe CContact

La classe *CContact* est utilisée dans le répertoire pour définir un contact. Elle se compose de onze variables de classes, permettant de définir précisément un contact. Vous trouverez ci-après son code source commenté, expliquant la signification de chaque champ, ainsi que la déclaration de ses différentes méthodes (getters, setters, et surdéfinition d'opérateurs).

---

```

/* CContact.h : Définition d'un objet CContact */
#define MAX 35          // constante
#define MAXTEL 11       // constante pour le numéro de téléphone
#include <ostream.h>     // pour les flux de sortie vers la console
#include <fstream.h>     // pour le flux de sortie vers les fichiers

class CContact {

private :
    /* Variables de classe */

    // Au moins une de ces deux variables doit être donnée pour créer l'objet CContact
    char _nom[MAX] ;      // nom du contact
    char _prenom[MAX] ;   // prénom du contact

    // Variables facultatives
    char _pseudo[MAX] ;   // pseudo du contact
    char _telephone[MAXTEL] ; // n° de téléphone du contact
    char _fax[MAX] ;      // n° de fax du contact
    char _naissance[MAX] ; // date de naissance du contact
    char _adresse[MAX] ;  // adresse du contact
    char _email[MAX] ;    // adresse e-mail du contact
    char _web[MAX] ;      // adresse web ou blog du contact
    char _type[MAX] ;     // choix entre plusieurs types prédéfinies
    char _note[MAX] ;     // infos complémentaires sur le contact

public :
    /* Constructeurs */
    CContact(void) ;      // constructeur par défaut
    CContact(char * nom, char * prenom) ; // constructeur

    /* Méthodes */

    // Les Getters (pour récupérer les champs privés de la classe)

```

---

```

char * getNom(char *);
char * getPrenom(char *);
...
char * getNote(char *);

// Les setters (pour définir les champs de la classe)
void setNom(char *);
void setPrenom(char *);
...
void setNote(char *);

// Surdéfinition de l'opérateur > pour l'ajout trié
bool operator > (CContact) ;

// Redéfinition de l'opérateur >> pour la saisie
friend ifstream& operator>>(ifstream&,CContact&);

// Redéfinition de l'opérateur << pour l'affichage
friend ostream& operator<<(ostream&,CContact&);

// Redéfinition de l'opérateur << pour la sauvegarde
friend ofstream& operator<<(ofstream&,CContact&);

};

```

Nous verrons plus en détails comment nous avons défini les différentes méthodes dans la partie 1.2 .

### 1.1.2 La classe **CRendezVous**

La classe *CRendezVous* définit les propriétés spécifiques à un rendez-vous. Les propriétés obligatoire étant : la date et l'heure de début du rendez-vous, et celles qui sont facultatives sont l'heure de fin du rendez-vous, le type (Réunion, Événement, Déplacement, Congés, ou Personnalisé), ou bien la description de celui-ci. La classe *CRendezVous* utilisant les classes *CDate* et *CHeure* celle possède donc une gestion assez poussée des dates et des heures. En effet à partir du string fournit par l'utilisateur ces classes vont extraire le jour, le mois, l'année, l'heure et les minutes puis effectués plusieurs vérifications de ces informations afin de ne pas enregistrer des données erronés.

### 1.1.3 La classe **CPropriétaire**

Nous avons défini une classe *CPropriétaire* pour gérer les paramètres concernant l'utilisateur du Pda, ainsi que ses préférences sur les couleurs de l'interface graphique. Cette classe est utilisée par la classe de gestion du menu Système (*CSysteme*) que nous verrons un peu plus loin, ainsi que par la classe principale (*CPda*) pour la connexion par login/mot de passe. Cette classe comporte quatorze champs listés ci-après :

- `mot_de_passe` : définit le mot de passe à entrer pour utiliser le Pda.
- `pseudo` : définit son pseudo (= login).
- `nom` : définit son nom.
- `prenom` : définit son prénom.

- `adresse1` : définit sa première adresse.
- `adresse2` : définit sa deuxième adresse.
- `telephoneFixe` : définit son numéro de téléphone fixe.
- `telephonePortable` : définit son nom numéro de téléphone portable.
- `fax` : définit son numéro de fax.
- `email` : définit son adresse e-mail.
- `web` : définit son adresse web.
- `note` : définit des informations que l'utilisateur aura envie de préciser.
- `couleurFond` : définit la couleur de fond du programme.
- `couleurTexte` : définit la couleur du texte du programme.

### 1.1.4 La classe *CPenseBete*

La classe *CPenseBete* nous sert à définir un pense bête pour la fonctionnalité Aide-mémoire présente dans les petits utilitaires. Cette classe propose 3 variables listées ci-après :

- `Titre` : définit le titre du pense-bête.
- `Description` : définit la description du pense-bête.
- `Type` : définit un type de pense bête parmi (Course, Travail, Idée, Personnalisé) le choix de l'option Personnalisée permet de définir un nouveau type de Pense-Bête.

## 1.2 Application des concepts du cours pour les objets de base

Nous allons voir dans cette partie comment nous avons appliqué les différents concepts appris en cours, tels que les getters, les setters, les fonctions amies et la surcharge d'opérateurs pour les objets de bases. Pour cela nous utiliserons le code source de la classe *CContact*, les autres classes fonctionnant sur le même principe.

### 1.1.1 Les Constructeurs

Pour chaque classe nous avons défini au minimum deux constructeurs, un par défaut initialisant toutes les variables de la classe, l'autre prenant un certain nombre de paramètre pour construire rapidement un objet sans passer par les setters.

Ainsi, pour la classe *CContact*, nous avons défini deux constructeurs :

```
CContact :: CContact(void) {                // constructeur par défaut
    strcpy(_nom, "");                        // on initialise toutes les variables
    strcpy(_prenom, "");
    ...
    strcpy(_note, "");
}

CContact :: CContact(char * nom, char * prenom) {
    strcpy(_nom, nom);                      // on définit le champ _nom au nom voulu
    strcpy(_prenom, prenom);                // on définit le champ _prenom au prénom voulu
    strcpy(_pseudo, "");                    // on initialise les autres variables...
    ...
    strcpy(_note, "");
}
```

### 1.1.2 Les Getters et setters

Voyons maintenant les méthodes de saisie et de récupération de variable de classe.

```
// Les Getters
char * CContact :: getNom(char * nom) { strcpy(nom,_nom); return nom; }
char * CContact :: getPrenom(char * prenom){strcpy(prenom,_prenom); return prenom; }
...
char * CContact :: getNote(char * note) { strcpy(note,_note); return note; }

// Les Setters
void CContact :: setNom(char * nom) { strcpy(_nom,nom) ; }
void CContact :: setPrenom(char * prenom) { strcpy(_prenom,prenom) ; }
...
void CContact :: setNote(char * note) { strcpy(_note,note) ; }
```

### 1.1.3 Les surdéfinitions d'opérateurs

Afin de simplifier l'utilisation des objets de bases, nous leur avons surdéfini plusieurs opérateurs.

Premièrement nous avons surdéfini l'opérateur << pour l'affichage de l'objet. Voyez ci-après cette méthode pour l'objet *CContact* :

```
// Redéfinition de l'opérateur << pour l'affichage
ostream& operator<< (ostream& o,CContact c) {
    char tmp[MAX];

    if(strcmp(c.getNom(tmp),"") != 0) cout << "Nom : " << c.getNom(tmp) ;
    if(strcmp(c.getPrenom(tmp),"") != 0) cout << "Prenom : " << c.getPrenom(tmp) ;
    ...
    if(strcmp(c.getNote(tmp),"") != 0) cout<<"Note ou commentaires:"<< c.getNote(tmp) ;

    return o;
}
```

La condition qui précède l'affichage de chaque champ nous permet d'afficher seulement les champs non vides.

Ensuite nous avons surdéfini le même opérateur mais cette fois ci pour la sauvegarde :

```
// Redéfinition de l'opérateur << pour la sauvegarde
ofstream& operator<< (ofstream& sortie,CContact c) {
    sortie << "\n" << c._nom ;
    sortie << "\n" << c._prenom ;
    ...
    sortie << "\n" << c._note ;

    return sortie;
}
```

De plus, nous avons surdéfini l'opérateur >> pour le chargement des objets, tel que vous pouvez le voir ci-après :

```
// Redéfinition de l'opérateur>> pour le chargement
ifstream& operator>> (ifstream& entree,CContact& c) {

    char n[MAX], p[MAX], t[MAXTEL], ps[MAX], tel[MAXTEL], f[MAX], na[MAX],
    ad[MAX], e[MAX], w[MAX], note[MAX] ;
```

```
// Pour chaque contact on récupère l'ensemble de ces champs
entree.getline(n,MAX);
entree.getline(p,MAX);
...
entree.getline(note,MAX);

// Puis nous définissons le contact grâce au setters
c.setNom(n);
c.setPrenom(p);
...
c.setNote(note);

return entree;
}
```

Enfin, nous avons surdéfini l'opérateur > qui permet un ajout trié de l'objet dans la liste :

```
// Surdéfinition de l'opérateur > pour l'ajout trié
bool CContact :: operator> (CContact c) {
    char tmp1[255],tmp2[255],tmp3[255];
    // on recopie le nom dans la variable temporaire tmp1
    strcpy(tmp1,_nom);

    // on ajoute a tmp1 (qui vaut le nom) le prénom pour trier par nom et par prénom
    char * mot1 = strcat(tmp1,_prenom);
    char * mot2 = strcat( c.getNom(tmp2),c.getPrenom(tmp3) );

    // on passe les variables en majuscule pour ignorer la casse lors de la comparaison
    strupr(mot1);
    strupr(mot2);
    return (strcmp(mot1,mot2)>0);
}
```

## 2. Les structures de données

Dès le début de la conception de notre projet nous nous sommes rapidement rendu compte que nous aurons besoin d'une liste (*CListe*) pour gérer nos structures de données telles que la liste de contacts, de rendez-vous, de pense-bêtes, et de notes. Nous avons alors programmé une liste générique doublement chaînée gérant les exceptions. Puis voulant apporter des méthodes de recherches avancées pour le répertoire et l'agenda nous avons défini deux listes héritières de *CListe* (respectivement *CListeContact* et *CListeRendezVous*). Ce choix de hiérarchie des classes a permis à nos classes de gestion des menus de se contenter de gérer les menus sans avoir à intervenir dans la liste pour les méthodes de recherche.

### 2.1 La Liste générique

Vous trouverez ci-dessous, le code source commenté de la déclaration des méthodes de la liste générique :

```
/* Méthodes de la classe */
// Les méthodes d'ajouts
void ajoutTete (Noeud<T> *) ;
void ajoutQueue (Noeud<T> *) ;
void ajoutAvtPos (Noeud<T> *,Noeud<T> *) ;
```

```
// ajoute un objet sur la position i (peut lancer 'Exception')
void ajout(T,int i);
// ajoute de manière triée un objet (grâce à la surdéfinition de l'opérateur >)
void ajoutTrie(T);

// Les méthodes de suppression
void supprTete(void); // peut lancer 'Exception'
void supprQueue(void); // peut lancer 'Exception'
void supprPos (Noeud<T> *); // peut lancer 'Exception'
void suppr(int i) ; // Supprime le Noeud de la ième position, peut lancer
// 'Exception'

// Renvoie le Noeud qui est à la ième position
Noeud<T> * getNoeud(int n) ;

// Renvoie l'objet du Noeud qui est à la ième position peut lancer 'Exception'
T get(int n) ;
void afficher(bool); // affiche tous les éléments de la liste
bool estVide(); // renvoie true si la liste est vide
int taille() ; // renvoie la taille de la liste
```

Comme vous avez pu le constater, la liste possède une méthode *ajout()* qui permet d'ajouter à n'importe quelle position et une méthode *ajoutTrie()* qui ajoute de manière triée les objets qui ont surdéfini l'opérateur « > ».

Voyons maintenant le code source commenté de ces deux méthodes.

```
template <class T> // pour rendre la liste générique
void CListe<T> :: ajout(T nv,int n) {
    Noeud<T> * nouv = new Noeud<T>(nv);
    if (n<0 || n>nbe) // on vérifie si la position est valide...
        throw Exception(n,1); // ... on lance une exception
    else { // si la position est valide...
        if (n==0) ajoutTete(nouv); // si la position est 0 on ajout en tête
        else if (n==nbe-1) ajoutQueue(nouv); // on ajoute en queue
        else { // sinon on ajoute en milieu de liste
            Noeud<T> * tmp = getNoeud(n) ;
            ajoutAvtPos(nouv,tmp);
        }
    }
}
```

Code source commenté de la méthode *ajoutTrie()* :

```
template <class T>
void CListe<T> :: ajoutTrie(T nouv) {
    Noeud<T> * nv = new Noeud<T>(nouv); // on transforme l'objet en Noeud
    Noeud<T> * tmp = _tete;
    bool trouve= false;

    if (estVide()) // si la liste est vide on ajout en tête
        ajoutTete(nv);

    else { //sinon on se déplace dans la liste...
        while( !trouve && (tmp != NULL) ) { //jusqu'à trouver un objet supérieur
            if (tmp->getObjet() > nouv)
                trouve=true;
            else
                tmp=tmp->getSuiv();
        }
    }
}
```

```

        if (trouve) // si on a trouvé un objet plus grand on l'ajoute juste avant
            ajoutAvtPos(nv,tmp);
        else // sinon on l'ajoute en queue
            ajoutQueue(nv);
    }
}

```

Comme vous l'avez sans doute remarqué nous avons eu recours à une classe d'Exception (*CException*) pour gérer les tentatives de débordement de la liste. Nous allons donc voir ici, un peu plus en détails cette classe.

*CException* comporte un constructeur avec deux paramètres qui sont :

- `int _ind` : indice qui est à l'origine du débordement
- `int _raison` : Cet entier correspond à la raison du débordement. Ainsi, selon le chiffre indiqué on affiche le message d'erreur correspondant.

Voici son code source :

```

Exception :: Exception(int i,int r) {
    _ind = i ;
    _raison=r;
}
/* Getter */
int Exception :: getInd() {
    return _ind ;
}
/* Méthode */
void Exception :: message() {
    switch(_raison) {
        case 1 :
            cout << "Aucun noeud n'est a la position : " << _ind << endl ;
            break;
        case 2 :
            cout << "La file est vide.";
            break;
    }
}
}

```

Grâce à cette classe toutes les méthodes de la liste qui ont un risque de débordement peuvent lancer une exception.

## 2.2 Les classes héritières de *CListe*

Pour Cortex2K5 nous avons voulu apporter des méthodes avancées de recherche sur les contacts et sur les rendez-vous. Nous aurions pu définir ses méthodes dans les classes de gestion des menus, cependant nous avons voulu suivre au maximum les principes de la programmation orientée objet, en respectant au mieux l'encapsulation des données. Ainsi, nous avons créé deux nouvelles classes *CListeContact* et *CListeRendezVous* qui apportent à *CListe* de nouvelles méthodes de recherche ainsi qu'une méthode de statistique sur contacts (pour la classe *CListeContact*) grâce à un autre concept vu en cours : l'Héritage.

### 2.2.1 *CListeContact*

*CListeContact* apporte donc les méthodes de recherches suivantes :

- Une première méthode *rechercher()*, qui prend en paramètre un `char*` et qui effectue une recherche sur le nom et le prénom.
- Une deuxième méthode *rechercher()* (définie par surcharge), qui prend en paramètre le nom ET le prénom pour un résultat plus précis de la recherche
- *rechercherPseudo()*, qui effectue une recherche sur le pseudo des contacts (si ce champ a été défini).
- *rechercherType()*, qui effectue une recherche sur le type des contacts (« Amie », « Colleague », « Famille », ou tout autre type défini par l'utilisateur)
- *rechercherTelephone()*, qui effectue une recherche sur le numéro de téléphone du contact.

Héritant de *CListe* la déclaration de son entête est la suivante :

```
class CListeContact : public CListe<Ccontact>
```

Cette classe possède également une fonction *statistique()* qui donne les différents pourcentages sur le type des contacts.

### 2.2.2 CListeRendezVous

La classe *CListeRendezVous* est très similaire à la classe *CListeContact* puisqu'elle gère une liste d'objets, ici des objets *CRendezVous*. Ainsi nous avons créé des méthodes de recherche, nous pouvons donc retrouver un rendez-vous en effectuant une recherche par date (*rechercherParDate()*) et une recherche par type (*rechercherParType()*). Toutes les autres fonctionnalités suivent la même logique que pour la classe *CListeContact* nous ne les réexposerons donc pas.

## 3. Le système de sauvegarde

La gestion de la sauvegarde se réalise de manière totalement transparente pour l'utilisateur. Ainsi, à chaque ajout dans le répertoire et lorsque l'utilisateur quitte le programme la liste est sauvegardée automatiquement. De même, le chargement s'effectue au lancement de l'application, conformément à ce qu'il avait été précisé dans le sujet.

Vous trouverez ci-dessous le code source commenté de la méthode de sauvegarde de la liste de contact :

```
void CListeContact :: sauvegarder(void) {
// si la liste n'est pas vide
    if(!estVide()) {
        Noeud<CContact> * tmp ;
// on ouvre le fichier en écriture
        ofstream sortie("ListeContact.dat");
        tmp = _tete ;
// tant qu'il y a des Contacts dans la liste on les sauvegarde
        while(tmp != NULL ) {
            CContact c ;
            c = tmp->getObjet();
// la surdéfinition de l'opérateur << pour la sauvegarde rend le code plus lisible
            sortie << c;
            tmp = tmp->getSuiv();
        }
        sortie.close();
    }
}
```



```

    }
}

```

Voyons maintenant le code sources commenté de la méthode de chargement :

```

void CListeContact :: charger(void) {
    // ouverture en lecture du fichier
    ifstream entree("ListeContact.dat");
    char n[MAX];
    // si l'ouverture s'est mal passé
    if (!entree) {
        cout << "Impossible de lire le fichier "<<endl;
    }
    // sinon on saisi chaque contacts un par un
    else {
        entree.getline(n,MAX);
        while ( entree.eof()==0 ) {
            CContact c;
            // grâce a la surdéfinition de l'opérateur >> pour le chargement des
            //contacts la création de la liste est très simple
            entree >> c;
            ajoutTrie(c);
        }
        entree.close();
    }
}

```

## 4. Problèmes rencontrés

Le passage Java à C++ n'est pas été simple. La gestion des pointeurs ne nous a pas laissés indifférents, puisqu'il a été absolument impossible de faire comme si ceux-ci n'existaient pas. Le compilateur Visual C++ ne nous a malheureusement pas aidés dans cette tâche. Les erreurs de programmation n'étant pas surlignées en temps réel (comme netBeans le fait pour le langage Java par exemple), nous a forcé à rester très concentrés lors de la conception des classes. Les explications de ces erreurs lors des différentes compilations sont parfois peu claires et il est même arrivé que nous ne voyions pas les liens entre l'erreur affichée et la solution apportée pour la résoudre... De plus nous avons l'impression que beaucoup de temps a été perdu en recopiage de déclaration ceci étant dû aux fichiers en-tête. Il suffit d'imaginer vouloir ajouter un nouveau champ à une classe comme *CContact* pour se rendre compte à quel point cela peut être fastidieux. A ces problèmes dus à l'apprentissage d'un nouveau langage s'ajoute finalement un mauvais choix de répartition des tâches de notre part. L'énoncé spécifiait que le projet comprendrait essentiellement deux axes qui étaient de gérer un agenda d'une part et un répertoire d'autre part. Nous nous étions donc répartis les tâches simplement : l'un ferait l'agenda et l'autre le répertoire... Or il s'est avéré que l'implémentation de ces deux fonctionnalités était très similaire et nous avons finalement conçus deux choses qu'une seule personne aurait pu faire toute seule, ce qui aurait permis à l'autre personne de gérer tout le reste (pda, système, classes utilitaire, jeux, calculatrice perfectionnée etc.).

## Conclusion

La programmation en C++ est vraiment un métier. Après un travail de longue haleine nous avons pu finir le projet dans les temps, même si nous en avons souvent douté lors de sa conception. Nous avons, malgré tout, pu mettre la quasi totalité des fonctionnalités que nous souhaitions implémenter. La programmation d'un Pda qui, de prime abord, avait l'air motivante, s'est montrée assez frustrante du fait des limites imposées par une interface console. Cependant, il est évident que nous avons beaucoup progressé dans ce nouveau langage et dans l'utilisation de Visual C++ grâce à ce projet. Mais nous sommes finalement devenus plus fan de Java que du C à cause des différents problèmes évoqués. Enfin, tout les concepts vu en cours ont pu être mis à profit dans CorTeX 2K5, même les plus poussés comme la surdéfinition d'opérateur, l'héritage, et l'enregistrement de fichier, ce qui est déjà très intéressant pour l'avenir de nos futurs projets en C++.