OBJECTIFS DU COURS

Les objectifs poursuivis dans ce cours sont ceux de permettre aux étudiants de :

- ✓ bien faire asseoir la programmation comme un puissant outil informatique incontournable ;
- ✓ maitriser les concepts de programmation à bon escient ;
- ✓ avoir un esprit innovateur, de compréhension facile, d'analyse et d'optimisation des programmes informatiques ;
- ✓ maitriser les concepts de programmation orienté objet avec le langage C# ;
- ✓ développer la rapidité et l'efficience à la manière de programmer ;
- ✓ participer à certains concours de grande envergure tels que ceux de Microsoft (Imagine cup), google…
- ✓ développer des logiciels qui résolvent les problèmes des entreprises.

INTRODUCTION

L'art de programmer, c'est l'art de faire résoudre des problèmes par des machines. Il s'agit bien d'un art, au sens de l'artisan, qui passe par une longue période d'apprentissage et d'imitation.

Un programme est une suite finie d'instructions permettant de réaliser une ou plusieurs tâche(s), de résoudre un problème, de manipuler des données. Tout programme est écrit dans un langage qui, traduit par un compilateur ou un interpréteur, pourra ensuite être exécuté automatiquement par l'ordinateur. Dès 1967 on dénombrait 120 langages, dont seuls 15 étaient vraiment utilisés. Les programmeurs utilisent encore aujourd'hui des langages créés dans les années 50, période d'intense innovation.

Certains langages ont été conçus pour le calcul scientifique, d'autres pour la gestion des entreprises, d'autres enfin pour la formalisation du raisonnement ou le calcul algébrique. Il existe aussi des langages étroitement adaptés à une finalité technique précise.
Au tout début, dans les années 40, les programmeurs devaient écrire dans le langage machine de l'ordinateur. Son vocabulaire est constitué de nombres binaires qui représentent les adresses des mémoires et les codes des opérations. Mais ce langage est très pénible pour le programmeur.

L'assembleur, conçu en 1950, permet de coder les opérations en utilisant des caractères alphabétiques (ADD pour l'addition, SUB pour la soustraction etc.) et il traduit ces codes en langage machine. Néanmoins il était nécessaire de définir des langages encore plus commodes pour le programmeur, des langages dits « de haut niveau » (ils sont relativement faciles à apprendre et à utiliser, mais leur utilisation par la machine suppose une cascade de traductions).

Dans le domaine de la programmation, on utilise aujourd'hui des langages de programmation de haut niveau (facilement compréhensibles par le programmeur) par opposition aux langages de bas niveau (de type assembleur) qui sont plus orientés vers le langage machine.

Parmi les exemples de langages de haut niveau, on peut citer les langages C, C#, Java, Visual Basic etc...

Le premier langage « de haut niveau » fut Fortran (« Formula Translation ») conçu par John Backus à IBM en 1954. Ses instructions ressemblent à des formules mathématiques et il est bien adapté aux besoins des scientifiques, mais incommode pour les travaux peu mathématiques et notamment pour programmer des logiciels de gestion. IBM considérait Fortran comme un langage « propriétaire » qui devait être utilisé uniquement sur ses machines.

Algol (Algorithmic Oriented Language) a été développé en 1958 par un consortium européen pour concurrencer Fortran.

Le Cobol (« Common Business Oriented Language », développé en 1959 par un consortium comprenant le Department of Defense) était destiné aux logiciels de gestion. Le Cobol emploie des mots et une syntaxe proches de l'anglais courant.

D'autres langages encore plus commodes furent introduits ensuite : Basic (« Beginner's All-Purpose Symbolic Instruction Code », 1964) peut être rapidement maîtrisé par le profane ; il est utilisé dans les écoles, entreprises et ménages.

Pascal (1970), langage « structuré » conçu de façon à éviter les erreurs de programmation notamment en encourageant la modularité, sera largement utilisé par les pédagogues qui veulent donner aux étudiants une première formation à la programmation.

C (1972) est un langage de haut niveau, mais il peut aussi être utilisé comme un assembleur car il permet de programmer des instructions au plus près de la « physique » de la machine. Beaucoup de logiciels pour les entreprises seront écrits dans ce langage souple dont l'utilisation est pourtant dangereuse pour le débutant : comme il permet de tout faire, il comporte peu de « garde-fous ».

Certains langages de haut niveau sont adaptés à des applications précises : APT (« Automatically Programmed Tools ») pour le contrôle des machines outils numériques, GPSS (« General Purpose Simulation System ») pour la construction des modèles de simulation, LISP (« List Processing », créé par John MacCarthy au MIT en 1959) pour manipuler des symboles et des listes (suites de symboles) plutôt que des données. LISP sera souvent utilisé en

intelligence artificielle. Scheme est, parmi les dialectes de LISP, celui qui rassemble le plus de partisans. Perl (créé par Larry Wall en 1987) est un langage de commande commode dans le monde Unix et pour les serveurs Web.

Les langages de quatrième génération (4GL), utilisés surtout pour la gestion et l'interrogation des bases de données, seront encore plus proches du langage humain. On peut citer Focus, SQL (« Structured Query Language ») et dBASE.

Les langages objet (que l'on appelle souvent « langage orientés objet ») comme Simula (1969), Smalltalk (créé par Alan Kay au PARC de Xerox, 1980), C++ (créé par Bjarne Stroustrup aux Bell Labs, 1983) ou Java (créé par Scott Mc Nealy[5] chez Sun, 1995) permettent d'écrire des logiciels fondés sur des objets réutilisables, programmes rassemblant un petit nombre de données et de traitements et qui communiquent entre eux par des messages. La logique des langages objet est proche de celle de la simulation. L'évolution des langages objet est allée vers la simplicité et la sécurité.

Le mot « programmation » recouvre des activités très diverses : l'utilisateur individuel « programme », même s'il ne s'en rend pas compte, lorsqu'il utilise Excel et Word ; il peut aussi, s'il a un tempérament de bricoleur, faire de petits programmes en Pascal ou en Scheme : mais dans la plupart des cas ce bricolage n'ira pas loin même s'il est ingénieux. Les gros programmes sont écrits par des équipes de programmeurs spécialisés qui se partagent les tâches et utilisent souvent des générateurs de code (comme Rational Rose) pour la partie la plus mécanique du travail d'écriture. La différence entre le programme individuel et le gros programme est du même ordre que celle qui existe entre le travail (éventuellement très réussi) qu'un bricoleur bien équipé peut réaliser à domicile et la construction d'une automobile ou d'un avion, qui suppose la maîtrise d'un ensemble de techniques et des mises au point dont seule une grosse entreprise peut être capable.

C# (prononcez C Sharp) est un langage récent. Il a été disponible en versions beta successives depuis l'année 2000 avant d'être officiellement disponible en février 2002 en même temps que la plate-forme .NET 1.0 de Microsoft à laquelle il est lié. C# ne peut fonctionner qu'avec cet environnement d'exécution. Celui-ci rend disponible aux programmes qui

s'exécutent en son sein un ensemble très important de classes. Le C# est un langage Orienté Objet.

Il est néanmoins nécessaire d'opérer une opération de traduction du langage de haut niveau vers le langage machine (binaire). On distingue deux types de traducteurs :

- ✓ l'interpréteur qui traduit les programmes instruction par instruction dans le cadre d'une interaction continue avec l'utilisateur.
- ✓ le compilateur qui traduit les programmes dans leur ensemble : tout le programme doit être fourni en bloc au compilateur pour la traduction.

Il en résulte deux grands types de langages de programmation :

- ✓ **les langages interprétés** : Ce type de langage est donc adapté au développement rapide de prototypes (on peut tester immédiatement ce que l'on est en train de faire). On citera par exemple les langages QBasic, Scheme etc.
- ✓ **les langages compilés** : les sources du code est dans un premier temps passé dans un compilateur qui va générer, sauf erreur, un exécutable qui est un fichier binaire compréhensible par votre machine. Ce type de langage est donc adapté à la réalisation d'applications plus efficaces ou de plus grande envergure (il y a une optimisation plus globale et la traduction est effectuée une seule fois).

En outre, un langage compilé permet de diffuser les programmes sous forme binaire, sans pour autant imposer la diffusion les sources du code.
Quelques exemples de langages compilés : C, C# etc.

A noter que certains langages peuvent être à la fois compilés et interprétés, comme par exemple CAML.

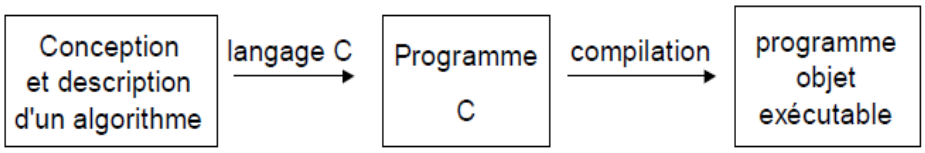


Figure 1 : Phases de développement d'un programme.

Le langage C est un langage de programmation procédural, l'idée principale de ce genre de langage étant de trouver les meilleurs algorithmes et d'en écrire des fonctions.
En fait, un programme C est organisé sous forme de fonctions et découpé en modules, ce qui autorise une très grande souplesse dans l'organisation de programmes complexes.

Voici sous forme d'un arbre généalogique l'origine du langage C :

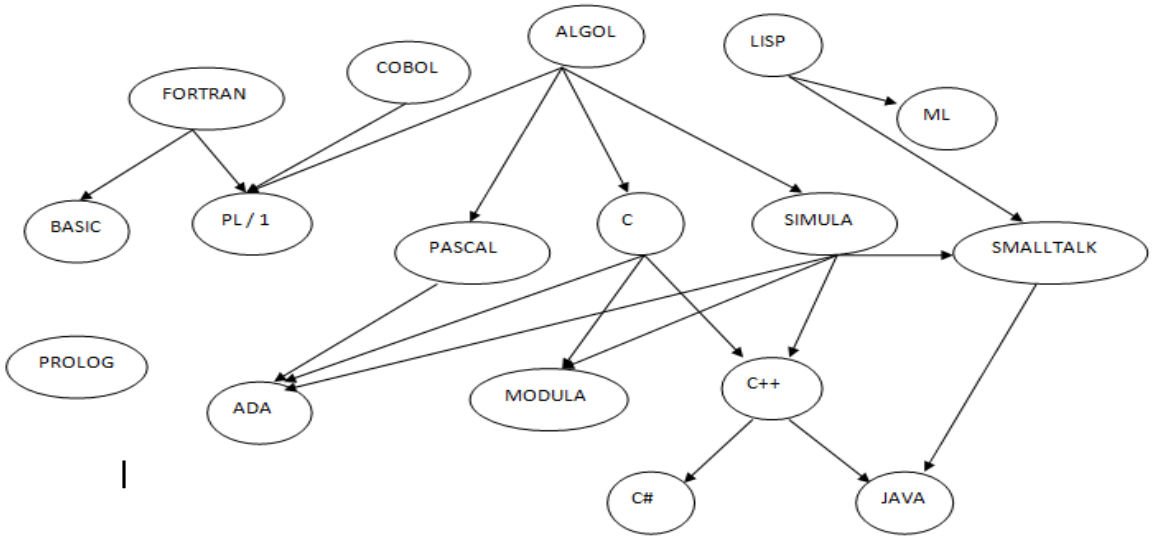


Figure 2 : Arbre généalogique de C et C#

CHAP. I. HISTORIQUE ET GENERALITES DU LANGAGE C

L'Histoire du langage C est intimement liée à celle du système d'exploitation UNIX. En 1965, **Ken Thompson**, de Bell Labs, développait un système d'exploitation qu'il baptisa MULTICS (Multiplexed Information and Computing System) afin de faire tourner un jeu qu'il avait créé, et qui donna naissance en 1970 au système d'exploitation UNICS (Uniplexed Information and Computing System) rapidement rebaptisé UNIX.

A l'époque, le seul langage qui permettait de développer un système d'exploitation était le langage d'assemblage. **Ken Thompson** développa alors un langage de plus haut niveau, le langage B (dont le nom provient de BCPL, un sous-ensemble du langage CPL, lui-même dérivé de l'Algol), pour faciliter l'écriture des systèmes d'exploitation.

C'était un langage faiblement typé (un langage non typé, par opposition à un langage typé, est un langage qui manipule les objets sous leur forme binaire, sans notion de type et trop dépendant du PDP-7 (la machine sur laquelle a été développé UNIX) pour permettre de porter UNIX sur d'autres machines.
Alors **Denis Ritchie** (qui fut, avec **Ken Thompson**, l'un des créateurs d'UNIX) et **Brian Kernighan** améliorèrent le langage B pour donner naissance au langage C. En 1973, UNIX fut réécrit entièrement en langage C. Pendant 5 ans, le langage C fut limité à l'usage interne de Bell jusqu'au jour où **Brian Kernighan** et **Denis Ritchie** publièrent une première définition du langage dans un ouvrage intitulé **The C Programming Language**.

Ce fut le début d'une révolution dans le monde de l'informatique. Grâce à sa puissance, le langage C devint rapidement très populaire et en 1983, l'ANSI (American National Standards Institute) décida de le normaliser en ajoutant également quelques modifications et améliorations, ce qui donna naissance en 1989 au langage tel que nous le connaissons aujourd'hui.

Les caractéristiques du langage C sont les suivants :

- ✓ **Universalité** : langage de programmation par excellence, le C n'est pas confiné à un domaine particulier d'applications. Il peut être utilisé aussi bien pour l'écriture de système d'exploitation que de programmes scientifiques ou de gestion, de logiciels modernes, de bases de données, de compilateurs, assembleurs ou interpréteurs, ...
- ✓ **Souplesse** : c'est un langage concis, très expressif, et les programmes écrits dans ce langage sont très compacts grâce à un jeu d'opérateurs puissant.
- ✓ **Puissance** : le C est un langage de haut niveau mais qui permet d'effectuer des opérations de bas niveau et d'accéder aux fonctionnalités du système, ce qui est la plupart du temps impossible dans les autres langages de haut niveau.
- ✓ **Portabilité** : c'est un langage qui ne dépend d'aucune plateforme matérielle ou logicielle. Le C permet en outre d'écrire des programmes portables, c'est-à-dire qui pourront être compilés sur n'importe quelle plateforme sans aucune modification.

De plus, sa popularité mais surtout l'élégance des programmes écrits en C est telle que sa syntaxe a influencé de nombreux langages dont C++ (qui est considéré comme un sur ensemble du C), JavaScript, Java, PHP et C#.

Le langage C est donc un langage compilé (même s'il existe des interpréteurs plus ou moins expérimentaux). Ainsi, un programme C est décrit par un fichier texte, appelé fichier source. La compilation se décompose en fait en 4 phases successives :

- ✓ Le traitement par le préprocesseur : le fichier source (portant l'extension.c) est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers sources, compilation conditionnelle ...).
- ✓ La compilation : la compilation proprement dite traduit le fichier généré par le préprocesseur (d'extension .i) en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.
- ✓ L'assemblage : cette opération transforme le code assembleur (extension.s) en un fichier binaire, c'est à dire en instructions directement compréhensibles par le processeur. Généralement, la

compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé fichier objet (et porte l'extension .o). Les fichiers objets correspondant aux librairies précompilées ont pour suffixe .a.

- ✓ L'édition de liens : un programme est souvent séparé en plusieurs fichiers sources, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standards déjà ´écrites. Une fois chaque fichier de code source assemble, il faut donc lier entre eux les différents fichiers objets. L'´édition de liens produit alors un fichier dit exécutable (a.out par défaut).

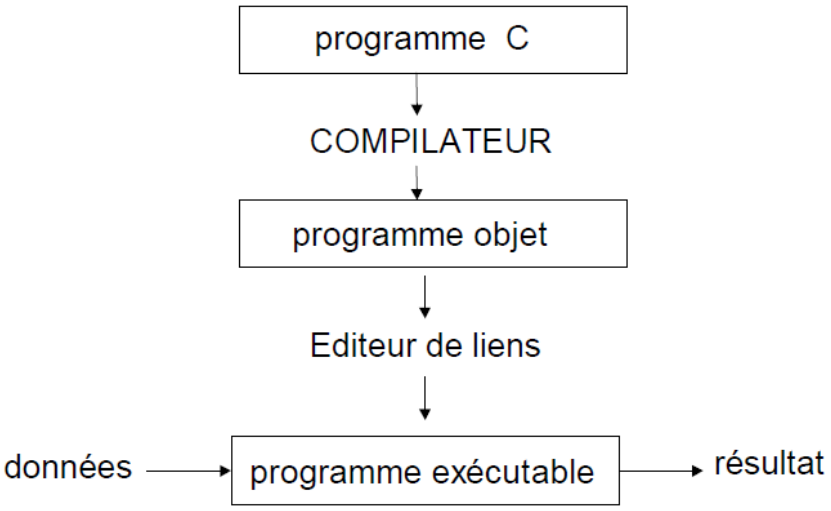


Figure 3 : Compilation d'un programme en C.

CHAP. II. BIBLIOTHEQUES STANDARDS

Le nom et la signature (le « prototype » en C) de chaque fonction sont décrits dans des en-têtes qui sont inclus dans le code source, tandis que le code objet des fonctions est séparée dans une bibliothèque logicielle, qui ne sera liée au reste du programme qu'au moment de l'édition de liens. Le nom et l'espace de noms des en-têtes sont devenus communs. Le plus souvent, chaque en-tête constitue un fichier séparé, mais l'organisation des implémentations reste diverse. La bibliothèque standard était initialement fournie avec le compilateur, mais aujourd'hui elle fait souvent partie du système d'exploitation. Cependant, le compilateur sait toujours où se trouvent ces fichiers, donc il est inutile de le lui préciser.

Sous Linux, c'est généralement la bibliothèque glibc qui est utilisée ; sous Windows, c'est généralement la bibliothèque standard de Visual C++, dénommée MSVCR*.dll, car elle est disponible indépendamment du compilateur C utilisé. Comme les compilateurs C offrent souvent plus de fonctionnalités que celles spécifiées par les normes C ISO et POSIX, une bibliothèque standard fournie avec un compilateur spécifique est peu compatible avec les bibliothèques standards des autres compilateurs pour les fonctions qui ne sont pas normalisées.

L'expérience a montré que la plus grande partie de la bibliothèque standard C a été bien conçue. La fonction de lecture de ligne gets et l'utilisation de scanf pour lire des chaînes en entrée sont la source de beaucoup de débordements de tampon, et la plupart des guides de programmation recommandent d'en éviter l'usage. Une autre vieillerie est strtok, une fonction conçue comme un analyseur lexical primitif mais qui est assez « fragile » et difficile à utiliser.

Le langage C primitif ne fournissait pas de fonctionnalités incorporées comme les opérations d'entrées/sorties (au contraire des langages traditionnels comme Pascal et Fortran). Avec le temps, les communautés d'utilisateurs de C ont partagé des idées et des implémentations de ce que nous appelons maintenant la bibliothèque standard de C pour fournir ces fonctionnalités.

Beaucoup de ces idées ont été incorporées dans la définition normalisée du langage de programmation C.

En comparaison avec d'autres langages (par exemple Java), la bibliothèque standard C de la norme ISO est minuscule. Elle fournit un jeu élémentaire de fonctions mathématiques, de manipulation de chaînes de caractères, de conversion de types, et d'entrée/sortie maniant les fichiers et les terminaux.

Voici la liste des quelques bibliothèques du langage C :

assert.h : Contient la macro assert, utilisée pour aider à détecter des incohérences de données et d'autres types de bogues dans les versions de débogage d'un programme.

complex.h : Pour manipuler les nombres complexes

errno.h : Ensemble (ou le plus souvent sous-ensemble) des codes d'erreurs renvoyés par les fonctions de la bibliothèque standard au travers de la variable errno.

fenv.h : Pour contrôler l'environnement en virgule flottante (floating-point)

float.h : Contient des constantes qui spécifient les propriétés des nombres en virgule flottante qui dépendent de l'implémentation, telles que la différence minimale entre deux nombres en virgule flottante différents.

inttypes.h : Pour des conversions précises entre types entiers.

iso646.h : Pour programmer avec le jeu de caractères ISO 646.

limits.h : Contient des constantes qui spécifient les propriétés des types entiers qui dépendent de l'implémentation.

locale.h : Pour s'adapter aux différentes conventions culturelles.

math.h : Pour calculer des fonctions mathématiques courantes.

setjmp.h : Pour exécuter des instructions goto non locales.

signal.h : Pour contrôler les *signaux* (conditions exceptionnelles demandant un traitement immédiat, par exemple signal de l'utilisateur).

stdarg.h : Pour créer des fonctions avec un nombre variable d'arguments.

stdbool.h : Pour avoir une sorte de type booléen (introduit par C99).

stddef.h : Définit plusieurs types et macros utiles, comme NULL.

stdint.h : Définit divers types d'entiers, c'est un sous-ensemble de inttypes.h.

stdio.h : Fournit les capacités centrales d'entrée/sortie du langage C, comme la fonction printf.

tgmath.h : Pour des opérations mathématiques sur des types génériques.

time.h : Pour convertir entre différents formats de date et d'heure.

wchar.h : Pour manipuler les caractères larges (wide char), nécessaire pour supporter un grand nombre de langues et singulièrement Unicode.

wctype.h : Pour classifier les caractères larges (introduit par Amd.1).

2.1. Utilisation des bibliothèques de fonctions

La pratique en C exige l'utilisation de bibliothèques de fonctions. Ces bibliothèques sont disponibles dans leur forme précompilée (extension: **.LIB**). Pour pouvoir les utiliser, il faut inclure des fichiers en-tête (*header files* **-** extension **.H**) dans nos programmes. Ces fichiers contiennent des '*prototypes*' des fonctions définies dans les bibliothèques et créent un lien entre les fonctions précompilées et nos programmes.

#include

L'instruction ***#include*** insère les fichiers en-tête indiqués comme arguments dans le texte du programme au moment de la compilation.

Identification des fichiers

Lors de la programmation en Borland C, nous travaillons donc avec différents types de fichiers qui sont identifiés par leurs extensions:

- ✓ .**C** : fichiers source
- ✓ .**OBJ** : fichiers compilés (version objet)
- ✓ .**EXE** : fichiers compilés et liés (version exécutable)

- ✓ .**LIB** : bibliothèques des fonctions précompilées
- ✓ .**H** : fichiers en-tête

Exemple

Nous voulons écrire un programme qui fait appel à des fonctions mathématiques et des fonctions graphiques prédéfinies. Pour pouvoir utiliser ces fonctions, le programme a besoin des bibliothèques:

MATHS.LIB et GRAPHICS.LIB

Nous devons donc inclure les fichiers en-tête correspondants dans le code source de notre programme à l'aide des instructions:

#include <math.h>

#include <graphics.h>

Après la compilation, les fonctions précompilées des bibliothèques seront ajoutées à notre programme pour former une version exécutable du programme.

Il est à noter que la bibliothèque de fonctions **graphics.h** est spécifique aux fonctionnalités du PC et n'est pas incluse dans le standard ANSI-C.

CHAP. III. LES BASES DE LA PROGRAMMATION EN C

Lorsque l'on désire créer un programme répondant à un cahier des charges bien défini (condition préalable évidement nécessaire), il faut déterminer quelles données il va falloir traiter, et comment les traiter. La première étape est donc de choisir comment représenter en mémoire ces données, et si plusieurs possibilités sont envisageables, choisir la plus appropriée aux traitements qu'il faudra effectuer (c'est à dire celle pour laquelle les algorithmes seront les plus faciles à mettre en œuvre).

Une fois ce modèle défini, le programme doit être écrit de manière structurée, c'est à dire être décomposé en petites entités (sous programmes, fonctions en C), réalisant chacune une tâche bien définie, en ayant bien défini quelles sont les données nécessaires en entrée du sous-programme, et quelles seront les données retournées en sortie du sous-programme (arguments ou dans certains cas variables globales). La réalisation pratique de la tâche doit ne dépendre que de ses entrées et sorties, et n'accéder à aucune autre variable (par contre elle peut utiliser pour son propre compte autant de variables locales que nécessaire).

Ceci permet d'éviter les effets de bord, qui rendent la recherche d'erreurs (débogage) presque impossible.

Le choix d'un modèle est capital : devoir le modifier une fois le programme bien avancé nécessite en général la réécriture complète du programme, alors que modifier certaines fonctionnalités du programme correspond à ajouter ou modifier des sous programmes sans modifier les autres. C'est un des intérêts de la programmation structurée. Par contre, pour pouvoir plus facilement modifier le modèle, il faut des structures de données hiérarchisées et évolutives (disponibles dans les langages orientés objets). Un autre avantage de la programmation structurée est la possibilité de créer dans un premier temps chaque sous-programme réalisant une tâche déterminée grâce à un algorithme simple, puis d'optimiser uniquement les sous−programmes souvent utilisés, ou demandant trop de temps de calcul, ou nécessitant trop de mémoire.

On n'optimise un programme (ou du moins certaines parties) que si l'on estime que son fonctionnement n'est pas acceptable (en temps ou en consommation de mémoire). On devra choisir un algorithme en fonction des conditions d'utilisation du programme.
A partir d'un moment, on ne peut plus optimiser en temps et en mémoire. Il faut alors choisir.
Par exemple, un résultat de calcul qui doit être réutilisé plus tard peut être mémorisé (gain de temps) ou on peut préférer refaire le calcul (gain de mémoire).

3.1. VARIABLES ET TYPES DES DONNEES

Les variables et les constantes sont les données principales qui peuvent être manipulées par un programme. Les déclarations introduisent les variables qui sont utilisées, fixent leur type et parfois aussi leur valeur de départ. Les opérateurs contrôlent les actions que subissent les valeurs des données. Pour produire de nouvelles valeurs, les variables et les constantes peuvent être combinées à l'aide des opérateurs dans des expressions.

Le type d'une donnée détermine l'ensemble des valeurs admissibles, le nombre d'octets à réserver en mémoire et l'ensemble des opérateurs qui peuvent y être appliqués.

La grande flexibilité de C nous permet d'utiliser des opérandes de différents types dans un même calcul. Cet avantage peut se transformer dans un terrible piège si nous ne prévoyons pas correctement les effets secondaires d'une telle opération (conversions de type automatiques, arrondissements, etc).

Au niveau du processeur, toutes les données sont représentées sous leur forme binaire et la notion de type n'a pas de sens. Cette notion n'a été introduite que par les langages de haut niveau dans le but de rendre les programmes plus rationnels et structurés. Mais même parmi les langages de haut niveau, on distingue ceux qui sont dits fortement typés (qui offrent une grande variété de types), comme le Pascal par exemple, de ceux qui sont dits faiblement ou moyennement typés (et plus proches de la machine) comme le C par exemple.

Le langage C ne dispose que de 4 types de base :

Type C	**Type Correspondant**
char	Caractère (entier de petite taille)
int	entier
float	Nombre flottant (réel) en simple précision
double	Nombre flottant (réel) en double précision

Type **Année**	**PDP 11** **1970**	**Intel 486** **1989**	**Sparc** **1993**	**Pentium** **1993**	**Alpha** **1994**
char	8 bits	8 bits	8 bits	8 bits	8 bits
short	16 bits	16 bits	16 bits	16 bits	16 bits
int	16 bits	16 bits	32 bits	32 bits	32 bits
long	32 bits	32 bits	32 bits	32 bits	64 bits
float	32 bits	32 bits	32 bits	32 bits	32 bits
double	64 bits	64 bits	64 bits	64 bits	64 bits
Long double	64 bits	64 bits	64 bits	64 bits	128 bits

Tableau 1 : Longueur des types de base sur quelques machines

Devant *char ou int*, on peut mettre le modificateur *signed ou unsigned* selon que l'on veut avoir un entier signé (par défaut) ou non signé.

Exemple :

```
char ch;
unsigned char c;
unsigned int n; /* ou tout simplement : unsigned n */
```

La plus petite valeur possible que l'on puisse affecter à une variable de type entier non signé est 0 alors que les entiers signés acceptent les valeurs négatives.

Devant le mot *int*, on peut mettre également short ou long auxquels cas on obtiendrait un entier court (*short int* ou tout simplement *short*) respectivement un entier long (*long int* ou tout simplement long).

Exemples
int n = 10, m = 5;
short a, b, c;
long x, y, z = -1;
unsigned long p = 2;

long peut être également mis devant *double*, le type résultant est alors *long double* (quadruple précision).

3.2. REGLE D'ECRITURE DES CONSTANTES LITTERALES

- ✓ Nombres entiers

Toute constante littérale « pure » de type entier (ex : 1, -3, 60, 40, -20, ...) est considérée par le langage comme étant de type int.
Pour expliciter qu'une constante littérale de type entier est de type unsigned, il suffit d'ajouter à la constante le suffixe u ou U. Par exemple : 2u, 30u, 40U, 50U, ...
De même, il suffit d'ajouter le suffixe l ou L pour expliciter qu'une constante littérale de type entier est de type long (on pourra utiliser le suffixe UL par exemple pour unsigned long).
Une constante littérale de type entier peut également s'écrire en octal (base 8) ou en hexadécimal (base 16). L'écriture en hexa est évidemment beaucoup plus utilisée. Une constante littérale écrite en octal doit être précédée de 0 (zéro). Par exemple : 012, 020, 030UL, etc.

Une constante littérale écrite en hexadécimal doit commencer par 0x (zéro x). Par exemple 0x30, 0x41, 0x61, 0xFFL, etc.

- ✓ Nombres flottants

Toute constante littérale « pure » de type flottant (ex : 0.5, -1.2, ...) est considérée comme étant de type double.
Le suffixe f ou F permet d'expliciter un float. Attention, 1f n'est pas valide car 1 est une constante entière. Par contre 1.0f est tout à fait correcte. Le suffixe l ou L permet d'expliciter un long double.

Une constante littérale de type flottant est constituée, dans cet ordre :

- d'un signe (+ ou -)
- d'une suite de chiffres décimaux : la partie entière
- d'un point : le séparateur décimal
- d'une suite de chiffres décimaux : la partie décimale
- d'une des deux lettres e ou E : symbole de la puissance de 10 (notation scientifique)
- d'un signe (+ ou -)
- d'une suite de chiffres décimaux : la puissance de 10

Par exemple, les constantes littérales suivantes représentent bien des nombres flottants : 1.0, -1.1f, 1.6E-19, 6.02e23L, 0.5 3e8

3.3. OPERATEURS ET EXPRESSION

Les expressions en langage C peuvent être constituées de variables, constantes, éléments de tableau et références à des fonctions combinés entre eux à l'aide d'opérateurs.

De façon générale, une expression est une combinaison d'opérateurs et d'opérandes dont le résultat est une valeur.

L'expression possède une valeur mais peut réaliser une affectation à une variable. Car les opérateurs d'affectation et d'incrémentation peuvent non seulement intervenir dans une expression (qui aura une valeur) mais agir sur le contenu des variables.

k = (i = 5) La valeur de l'expression (i = 5) est affectée á la valeur de k

En fait, les notions d'expression et d'instruction sont étroitement liées.

Quand une expression possède plusieurs opérateurs, l'ordre dans lequel les opérations sont effectuées est important.

3.3.1. Les opérateurs arithmétiques:

On distingue:

a) opérateurs arithmétiques binaires

+	addition
-	soustraction
/	division
*	multiplication
%	Reste de la division entière (modulo)

Les opérateurs binaires ne sont définis que sur deux opérandes ayant le même type : (int, long int, float, double,long double). Ils fournissent le même type que leurs opérandes et l'opérateur modulo (%) ne peut porter que sur des entiers.

b) Un opérateur unaire : est un operateur qui ne porte que sur un opérande.

Lorsque plusieurs opérateurs arithmétiques apparaissent dans une expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu.

L'ordre de priorité des opérateurs arithmétiques est le suivant:

-(opérateur unaire	Plus prioritaire
* / %	
+ -	Moins prioritaire

L'évaluation des expressions arithmétiques suit l'ordre de précédence des opérateurs.
Dans le cas où des opérateurs ont des priorités identiques, l'évaluation de l'expression se fait de gauche à droite.
L'utilisation des parenthèses dans une expression permet d'altérer les règles de priorité. Les sous expressions parenthèses sont évaluées en priorité.

c) Les conversions implicites dans un calcul d'expression

Une expression mixte est une expression, dans laquelle interviennent des opérandes de types différents. Dans une expression mixte, le compilateur met en place des instructions de conversion de la valeur d'un opérande pour obtenir une expression dont tous les opérandes ont le même type. Le résultat de la conversion sera exprimé dans le type de plus haute précision.

On distingue deux types de conversion :

- **la conversion d'ajustement de type**

Une conversion de type suit un certain ordre qui permet de ne pas dénaturer la valeur initiale.

int ---> long --->float ---> double ---> long double

Cet ordre permet de convertir par exemple, **int** en **long** ou **double** ou **long double** mais l'inverse n'est pas possible.

- **Les promotions numériques**

Les opérateurs arithmétiques ne sont pas définis pour le type **short** et **char**. Le langage C prévoit que toute valeur de l'un de ces types apparaissant dans une expression, est d'abord convertie en **int**. On parle alors de promotion numérique.

Cas du type char

La promotion numérique permet de considérer le code du caractère (sur 8 bits) comme la valeur de ce caractère. Dans ce cas, le langage C confond un caractère avec la valeur (entier) du code qui le représente.

La valeur entière associée à un caractère donné n'est pas le même sur toutes les machines.

3.3.2. Les opérateurs relationnels

Le langage C permet de comparer des expressions à l'aide d'opérateurs de comparaison.Le résultat de la comparaison est une valeur entière de valeur **1** (si le résultat est **vrai**) ou **0** (si le résultat est **faux**).

Cette expression faisant intervenir des opérateurs de comparaison, sera alors de type entier et donc pourra intervenir dans des calculs arithmétiques.

Les opérateurs relationnels sont :

<	Inferieur à
>	Supérieur à
<=	Inferieur ou égal à
>=	Supérieur ou égal à

==	Egal à
!=	Diffèrent de

Ils sont classés suivant cet ordre de priorité :

<,>,<=,>= (plus haute priorité)

= = et != (moins prioritaire)

Les opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques.
Une comparaison peut porter sur 2 caractères

a) cas de comparaison d'égalité

(L'existence d'une conversion de promotion numérique **char** --->**int** n'a guère d'influence).

char c1,c2;

c1 = = c2 est vraie si c1 et c2 ont la même valeur ou bien c1 et c2 contiennent des caractères de même code (le même caractère).

b) cas de comparaison d'inégalité

char c1,c2;

c1 < c2 est vraie si le code du caractère c1 a une valeur inférieure au code du caractère c2. Le résultat dépendra du codage employé.

3.3.3. Les opérateurs logiques

Le langage C dispose de 3 opérateurs logiques:

&&	ET logique
||	Ou
!	non

A priori, les opérateurs logiques ne portent que sur des opérandes qui sont eux-mêmes des expressions logiques. Mais, ils acceptent des opérandes numériques (types int et float) avec les règles de conversion implicites. Dans ce contexte, on considère que :

La valeur nulle (0) ne correspond à **faux et toute valeur non nulle** ne correspond à **vrai.**

Fonctionnement des opérateurs logiques dans C :

Opérande 1	**Opérateur**	**Opérande 2**	**résultat**
0	**&&**	0	0
0	**&&**	Valeur non nulle	0
Valeur non nulle	**&&**	0	0
Valeur non nulle	**&&**	Valeur non nulle	Valeur non nulle
0	**||**	0	0
0	**||**	Valeur non nulle	Valeur non nulle
Valeur non nulle	**||**	0	Valeur non nulle
Valeur non nulle	**||**	Valeur non nulle	Valeur non nulle
	!	0	Valeur non nulle
	!	Valeur non nulle	0

Les règles de priorité des opérateurs logiques

! Plus prioritaire

&&

|| Moins prioritaire

Les opérateurs arithmétiques sont plus prioritaires que les opérateurs logiques. L'opérateur ! (Not) a une priorité supérieure à celle de tous les opérateurs arithmétiques binaires et opérateurs relationnels.

3.3.4. Les opérateurs d'affectation

L'opérateur d'affectation permet de former des **expressions d'affectation**.

La partie gauche de l'opérateur d'affectation **« = »**doit être une **lvalue** (left value).

Lvalue est une expression à laquelle on peut affecter une valeur.

Les variables sont des lvalues.

La partie à droite de l'opérateur est une expression dont la valeur est affectée à la lvalue.

- La priorité de cet opérateur est inférieure à celle de tous les opérateurs arithmétiques et les opérateurs de comparaison.
- Si les opérandes sont de types différents, il y a **conversion systématique** de l'expression dans le type de la lvalue. Cette conversion peut entraîner à une dégradation de la valeur convertie.

Par exemple, une valeur de type float peut être tronquée (elle perd sa partie décimale) si elle est affectée à une lvalue de type entier.

3.3.5. Les opérateurs d'incrémentation et de décrémentation:

L'opérateur d'incrémentation noté ++ et celui de décrémentation noté - -

Ils ont pour effet d'incrémenter ou de décrémenter de 1 la valeur d'une variable (lvalue).

- **un opérateur de pré-décrémentation ou de pré-incrémentation** s'il est placé à gauche de la `` lvalue`` sur laquelle il porte. La valeur de l'expression décrémentation ou incrémentation est celle de la ``lvalue`` après décrémentation ou incrémentation.
- **un opérateur de post-décrémentation ou post-incrémentation** s'il est placé à droite de la `` lvalue`` sur laquelle il porte. La valeur de l'expression décrémentation ou incrémentation est celle de la ``lvalue`` avant décrémentation ou incrémentation.

Ces opérateurs sont de plus haute priorité que les opérateurs arithmétiques.

3.3.6. Les opérateurs d'affectation élargie:

D'une manière générale, C permet de condenser les affectations de la forme

lvalue = lvalue opérateur expression en **lvalue opérateur = expression**

Ceci concerne seulement les opérateurs arithmétiques.

Liste des opérateurs d'affectation élargie

+=, -=, *=, /=, %= (concernant les opérateurs arithmétiques).

3.3.7. L'opérateur Cast

Le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix à l'aide de l'opérateur **cast.**

- Sa priorité est élevée par rapport aux autres opérateurs.
- Toutes les conversions numériques sont réalisables par un opérateur **cast**.

3.3.8. L'opérateur conditionnel :

L'opérateur conditionnel est un opérateur ternaire mettant en relation trois expressions ou opérandes. Il évalue la première expression qui joue le rôle de condition :

- **si sa valeur est non nulle**, il y a évaluation du second opérande (expression) qui sera la valeur du résultat.
- **si sa valeur est nulle**, il y a évaluation du troisième opérande qui sera la valeur du résultat.

Sa priorité est faible, relativement aux autres opérateurs.

3.3.9. L'opérateur sizeof

Son emploi ressemble à celui d'une fonction. Il fournit la taille (en octet) de son paramètre. Cet opérateur peut être appliqué soit à une variable soit à un type.

3.4. STRUCTURES DE CONTROLE

Un programme ayant un fonctionnement linéaire, un comportement déterminé précisément et invariant, n'est généralement pas très utile ; certes, cela permet d'effectuer un grand nombre de fois une tâche répétitive, simplement en lançant le programme le nombre de fois voulu... Mais, généralement, il faut que le programme s'adapte à des cas particuliers, à des conditions, qu'il exécute certaines fois une portion de code, d'autres fois une autre portion de code, qu'il soit capable de répéter plusieurs fois la même tâche sur des données successives... Tout ceci nécessite ce qu'on appelle "structures de contrôle".

Le C propose plusieurs structures de contrôle différentes, qui nous permettent de nous adapter à tous les cas possibles. Il permet d'utiliser des conditions

(structures alternatives), des boucles (structures répétitives), des branchements conditionnels ou non, ...

Certaines de ces structures de contrôle sont quasiment toujours utilisées ; d'autres le sont moins. Certaines sont très appréciées, d'autres sont à éviter...

3.4.1. Structures conditionnelles

A. if

La forme la plus simple de structure conditionnelle est d'exécuter quelque chose dans le cas où une condition est vraie.

```
if(condition)
   Instruction à exécuter si la condition est vraie.
```

Par exemple, pour afficher "a vaut 5" dans le cas où la variable a contient effectivement la valeur 5, on utilisera le code suivant :

```
if(a == 5)
   printf("a vaut 5");
```

Notez que l'indentation (le fait que l'instruction à exécuter si la condition est vraie soit décalée vers la droite) n'influe en rien le comportement du programme : on peut tout à fait ne pas indenter. Cela dit, indenter correctement permet de rendre le code source beaucoup plus lisible implémenté en les regroupant à l'intérieur d'un bloc.

```
if(a == 5)
{
   clrscr();
   printf("a vaut 5");
   ngetchx();
}
```

Il est souvent nécessaire de pouvoir exécuter une série d'instructions dans le cas où une condition est vraie, et d'exécuter une autre série d'instructions dans le cas contraire...

```
if(condition)
   Instruction à exécuter si la condition est vraie.
else
   Instruction à exécuter si la condition est fausse.
```

On peut aussi être amené à vouloir gérer plusieurs cas particuliers, et un cas général, correspondant au fait qu'aucun des autres cas n'a été vérifié. Pour cela, nous emploierons la structure de contrôle dont la syntaxe est la suivante, et qui est la forme la plus complète de if :

```
if(condition1)
   Instruction à exécuter si la condition1 est vraie.
else if (condition2)
   Instruction à exécuter si la condition2 est vraie.
else if (condition3)
   Instruction à exécuter si la condition3 est vraie.
...
...
else
   Instruction à exécuter si les condtions 1, 2, et 3 sont toutes les trois fausses.
```

B. Opérateur (condition) ? (vrai) : (faux)

Le C fournit aux programmeurs un opérateur permettant d'obtenir une valeur si une condition est vraie, et une autre si la condition est fausse ; il s'agit de l'opérateur ?:, qui s'utilise comme ceci :

```
condition ? valeur1: valeur2
```

Et toute cette expression prend la valeur valeur1 si la condition est vraie, ou la valeur valeur2 si la condition est fausse.

Par exemple, pour affecter à une variable une valeur en fonction du résultat de l'évaluation d'une condition, on pourra utiliser ceci :

```
short a;
short result;
result = (a==10 ? 100 : 200);
```

Notons que cette écriture n'est que la forme concise de celle-ci :

```
short a;
short result;
if(a == 10)
   result = 100;
else
   result = 200;
```

Mais utiliser l'opérateur**?:** permet d'utiliser une valeur en fonction d'une condition là où on ne pourrait pas mettre un bloc if...else.

Par exemple, l'opérateur **?:** est extrêmement pratique dans ce genre de situation :

```
short a;
short result;
result = 2*(a==10 ? 100 : 200);
```

L'opérateur **?:** est, puisqu'il fonctionne avec trois opérandes, un opérateur "ternaire".
Cela dit, étant donné que c'est le seul opérateur de ce genre que fournit le C, on a généralement tendance à l'appeler "l'opérateur ternaire", plutôt que "expression conditionnelle".

3.4.2. Structures itératives

Le C présente trois types de structures de contrôle itératives, c'est-à-dire, de structures de contrôle permettant de réaliser ce qu'on appelle des boucles ; autrement dit, d'exécuter plusieurs fois une portion de code, généralement jusqu'à ce qu'une condition soit fausse.

Le plus grand danger que présentent les itératifs est que leur condition de sortie de boucle ne soit jamais fausse.

A. while

La première des itératives est la boucle "while", appelée "tant que" en français, lorsque l'on fait de l'algorithme.

Avec cette structure de contrôle, tant qu'une condition est vraie, les instructions lui correspondant sont exécutées.

```
while(condition)
   Instruction à effectuer tant que la condition est vraie.
```

Exemple

```
a = 0;
while(a <= 2)
{
   printf("a=%d\n", a);
   a++;
}
```

B. do... while

La seconde structure de boucle, que nous allons maintenant étudier, est "do...while", que l'on pourrait, en français, appeler "faire... tant que". Ici encore, les instructions constituant la boucle sont exécutées tant que la condition de boucle est vraie. Cela dit, contrairement à while, avec do...while, la condition est évaluée à la fin de la boucle ; cela signifie que les instructions correspondant au corps de la structure de contrôle seront toujours exécutées au moins une fois, même si la condition est toujours fausse !

```
do
   Instruction à exécuter tant que la condition est vraie.
while(condition);
```

Exemple

```
a = 0;
do
{
   printf("a=%d\n", a);
   a++;
}while(a <= 2);
```

C. for

La troisièmestructure de contrôle itérative est celle que l'on appelle boucle "for". Elle est généralement utilisée lorsque l'on veut répéter un nombre de fois connu une action.

for(initialisation ; condition ; opération sur la variable de boucle)
    Instruction à exécuter tant que la condition est vraie.

Les trois expressions que j'ai nommé initialisation, condition, et opération sur la variable de boucle sont toutes trois optionnelle ; si aucune condition n'est précisée, le compilateur supposera que la condition est toujours vraie. Les points-virgules, eux, par contre, sont obligatoires !

C'est-à-dire :

- ✓ Disposer d'une variable de boucle, qui sera considérée comme un compteur du nombre de fois dont on est passé dans la boucle.
- ✓ Initialiser cette variable, ce que l'on fait grâce à la première expression du for.
- ✓ Avoir une condition de boucle : une fois cette condition devenue fausse, on cessera de boucler. En règle générale, il est recommandé que cette condition porte sur la variable de compteur !
- ✓ Modifier la valeur de la variable de boucle utilisée comme compteur.

Par exemple,

```
for(a=0 ; a<=2 ; a++)
{
   printf("a=%d\n", a);
}
```

Le fonctionnement de ceci est tout simple : on initialise à 0 notre variable, on vérifie qu'elle est inférieure ou égale à deux, on affiche sa valeur, on l'incrémente, on vérifie qu'elle est inférieure ou égale à deux, on affiche sa valeur, on l'incrémente, ...

L'initialisation se fait une et une seule fois, au tout début, avant de rentrer dans la boucle ; la condition est évaluée en début de boucle, exactement comme pour while, et l'opération sur la variable de fin de boucle est effectuée en fin de boucle, avant de boucler.

Notez que la forme où on ne place ni initialisation, ni condition, ni opération, est souvent utilisée comme boucle infinie, dont il est possible de se sortir en utilisant certaines instructions d'altération de contrôle de boucle, que nous allons voir très bien ; pour illustrer ce propos, voici la syntaxe correspondant à une boucle infinie :

```
for ( ; ;)
   Instruction à exécuter sans cesse..
```

D. Instructions d'altération de contrôle de boucle

Le langage C propose plusieurs instructions qui permettent d'altérer le contrôle de boucles itératives, soit en forçant le programme à passer à l'itération suivante sans finir d'exécuter les instructions correspondant à celle qui est en cours, soit en forçant le programme à quitter la boucle, comme si la condition était fausse.

✓ Continue

L'instruction continue permet de passer au cycle suivant d'une boucle, sans exécuter les instructions restantes de l'itération en cours. Considérez l'exemple suivant :

```
for(a=0 ; a<=5 ; a++)
{
   if(a == 3)
      continue;
   printf("a=%d\n", a);
}
```

Lorsque a sera différent de 3, l'appel à printf permettra d'afficher sa valeur. Mais, lorsque a vaudra 3, on exécutera l'instruction while. On retournera immédiatement au début de la boucle, en incrémentant a au passage.

✓ Break

L'instruction break, elle, met fin au parcours de la boucle, sitôt qu'elle est rencontrée, comme si la condition d'itération était devenue fausse, mais sans même finir de parcourir les instructions correspondant au cycle en cours.

Exemple

```
for(a=0 ; a<=5 ; a++)
{
   if(a == 3)
      break;
   printf("a=%d\n", a);
}
```

Ce programme affichera la valeur de a lorsque a est inférieur à 3. Lorsque a vaudra 3, on exécutera l'instruction break... On quittera alors la boucle, sans même afficher la valeur 3, puisque l'appel à printf suit le break.

3.4.3. Structure conditionnelle particulière

Le langage C présente une structure conditionnelle particulière, le switch.

Cette structure est particulière dans le sens où elle ne permet que de comparer une variable à plusieurs valeurs, entières.

```
switch(nom_de_la_variable)
{
   case valeur_1:
         Instructions à exécuter dans le cas où la variable vaut valeur_1
      break;
   case valeur_2:
         Instructions à exécuter dans le cas où la variable vaut valeur_2
      break;
   default:
         Instructions à exécuter dans le cas où la variable vaut une valeur autre
que valeur_1 et valeur_2
      break;
}
```

Une structure switch peut avoir autant de case que vous le souhaitez. Le cas 'default' est optionnel : si vous le mettez, les instructions lui correspondant seront exécutées si la variable ne vaut aucune des valeurs précisées dans les autres cas ; si vous ne le mettez pas et que la variable est différente des valeurs précisées dans les autres cas, rien ne se passera

Exemple

```
short b = 20;
switch(b)
{
   case 5:
         printf("b vaut 5\n");
      break;
   case 10:
         printf("b vaut 10\n");
      break;
   default:
         printf("b ne vaut ni 5 ni 10\n");
      break;
}
```

3.4.4. Branchement inconditionnel

Un opérateur de branchement inconditionnel permet de "sauter" vers un autre endroit dans le programme, sans qu'il n'y ait de condition imposée par l'opérateur, au contraire des boucles ou des opérations conditionnelles, par exemple.

✓ **L'opérateur goto**

Lorsque l'on parle d'opérateurs de branchement inconditionnels, le premier qui vienne généralement à l'esprit des programmeurs est le goto. Il permet de brancher sur ce qu'on appelle une "étiquette" (un "label", en anglais), déclaré comme suit :

nom_de_l_etiquette:

C'est à dire un identifiant, le nom de l'étiquette, qui doit être conforme aux normes concernant les noms de variables, suivi d'un caractère deux-points.

Et l'instruction goto s'utilise de la manière suivante :

gotonom_de_l_etiquette_sur_laquelle_on_souhaite_brancher;

Notez cependant que goto ne peut brancher que sur une étiquette placée dans la même fonction que lui.

Exemple

```
    printf("blabla 1\n");
    goto plus_loin;
    printf("blabla 2\n");
plus_loin:
    printf("blabla 3\n");
```

✓ **L'opérateur return**

Cet opérateur permet de quitter la fonction dans laquelle on l'appelle. Si la fonction courante est la fonction _main, alors, return quittera le programme.

return;

3.5. ENTREES / SORTIES DEPUIS LE CLAVIER

Les **entrées/sorties** (**E/S**) ne font pas vraiment partie du langage C car ces opérations sont dépendantes du système. Cela signifie que pour réaliser des opérations d'entrée/sortie en C, il faut en principe passer par les fonctionnalités offertes par le système. Néanmoins sa bibliothèque standard est fournie avec des fonctions permettant d'effectuer de telles opérations afin de faciliter l'écriture de code portable. Les fonctions et types de données liées aux entrées/sorties sont principalement déclarés dans le fichier **stdio.h** (**st**an**d**ard **i**nput/**o**utput).

3.5.1. La fonction printf()

La fonction **printf** est utilisée pour transférer du texte, des valeurs de variables ou des résultats d'expressions vers le fichier de sortie standard *stdout* (par défaut l'écran).

printf("<format>",<Expr1>,<Expr2>, ...)

La partie "**<format>**" est en fait une chaîne de caractères qui peut contenir du texte, des séquences d'échappement et des spécificateurs de format.

Les spécificateurs de format commencent toujours par le symbole **%** et se terminent par un ou deux caractères qui indiquent le format d'impression.

Les spécificateurs de format impliquent une conversion d'un nombre en chaîne de caractères. Ils sont encore appelés *symboles de conversion*.

3.5.2. La fonction scanf()

La fonction **scanf** nous offre pratiquement les mêmes conversions que **printf**, mais en sens inverse.

scanf("<format>",<AdrVar1>,<AdrVar2>, ...)

La fonction **scanf** reçoit ses données à partir du fichier d'entrée standard *stdin* (par défaut le clavier).

- La chaîne de format détermine comment les données reçues doivent être interprétées.
- L'adresse d'une variable est indiquée par le nom de la variable précédé du signe **&**.

3.5.3. Écriture d'un caractère

La commande **putchar('a')** transfère le caractère *a* vers le fichier standard de sortie *stdout*. Les arguments de la fonction **putchar** sont ou bien des caractères c'est-à-dire des nombres entiers entre 0 et 255) ou bien le symbole **EOF** *(End Of File)*.

EOF est une constante définie dans <*stdio*> qui marque la fin d'un fichier. La commande **putchar(EOF);** est utilisée dans le cas où *stdout* est dévié vers un fichier.

Type de l'argument

Pour ne pas être confondue avec un caractère, la constante **EOF** doit nécessairement avoir une valeur qui sort du domaine des caractères (en général **EOF** a la valeur -1). Ainsi, les arguments de **putchar** sont par définition du type **int**et toutes les valeurs traitées par **putchar** (même celles du type **char**) sont d'abord converties en **int**.

3.5.4. Lecture d'un caractère

Une fonction plus souvent utilisée que **putchar** est la fonction **getchar**, qui lit le prochain caractère du fichier d'entrée standard *stdin*.

Type du résultat

Les valeurs retournées par **getchar** sont ou bien des caractères (0 - 255) ou bien le symbole **EOF**. Comme la valeur du symbole **EOF** sort du domaine des caractères, le type résultat de **getchar** est **int**. En général, **getchar** est utilisé dans une affectation:

```
int C;
  C = getchar();
```

getchar lit les données de la zone tampon de *stdin* et fournit les données seulement après confirmation par 'Enter'. La bibliothèque <*conio*> contient une fonction du nom **getch** qui fournit immédiatement le prochain caractère entré au clavier.

La fonction **getch** n'est pas compatible avec ANSI-C et elle peut seulement être utilisée sous MS-DOS.

3.6. FONCTIONS

On appelle *fonction* un sous-programme qui permet d'effectuer un ensemble d'instructions par simple appel de la fonction dans le corps du programme principal. Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, cela permet une simplicité du code et donc une taille de programme minimale. D'autre part, une fonction peut faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme...).

La plupart des langages de programmation nous permettent de subdiviser nos programmes en sous-programmes, fonctions ou procédures plus simples et plus compacts. A l'aide de ces structures nous pouvons *modulariser* nos programmes pour obtenir des solutions plus élégantes et plus efficientes.

Dans ce contexte, un *module* désigne une entité de données et d'instructions qui fournissent une solution à une partie bien définie d'un problème plus complexe. Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules ainsi reliés doit alors être capable de résoudre le problème global.

3.6.1. Avantages

Voici quelques avantages d'un programme modulaire:

- ✓ Meilleure lisibilité
- ✓ Diminution du risque d'erreurs
- ✓ Possibilité de tests sélectifs
- ✓ Dissimulation des méthodes

- ✓ Réutilisation de modules déjà existants
- ✓ Simplicité de l'entretien
- ✓ Favorisation du travail en équipe
- ✓ Hiérarchisation des modules

Un programme peut d'abord être résolu globalement au niveau du module principal. Les détails peuvent être reportés à des modules sous-ordonnés qui peuvent eux aussi être subdivisés en sous-modules et ainsi de suite. De cette façon, nous obtenons une *hiérarchie de modules*.

Les modules peuvent être développés en passant du haut vers le bas dans la hiérarchie *('top-down-development' - méthode du raffinement progressif)* ou bien en passant du bas vers le haut *('bottom-up-development')*.

En principe, la méthode du raffinement progressif est à préférer, mais en pratique, on aboutit souvent à une méthode hybride, qui construit un programme en considérant aussi bien le problème posé que les possibilités de l'ordinateur ciblé *('méthode du jo-jo')*.

3.6.2. La déclaration d'une fonction

Avant d'être utilisée, une fonction doit être définie car pour l'appeler dans le corps du programme il faut que le compilateur la connaisse, c'est-à-dire qu'il connaisse son nom, ses arguments et les instructions qu'elle contient. La définition d'une fonction s'appelle « *déclaration* ». La déclaration d'une fonction se fait selon la syntaxe suivante :

```
type_de_donnee Nom_De_La_Fonction(type1 argument1, type2 argument2, ...) {
liste d'instructions
}
```

- ✓ type_de_donnee représente le type de valeur que la fonction est sensée retourner (char, int, float...)

- Si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot-clé *void*
- Si aucun type de donnée n'est précisé alors le type *int* est pris par défaut

✓ Le nom de la fonction suit les mêmes règles que les noms de variables :
- le nom doit commencer par une lettre
- un nom de fonction peut comporter des lettres, des chiffres et les caractères _ et & (les espaces ne sont pas autorisés !)
- le nom de la fonction, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules)

✓ Les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes

✓ Il ne faut pas oublier de refermer les accolades

✓ Le nombre d'accolades ouvertes (fonction, boucles et autres structures) doit être égal au nombre d'accolades fermées !

✓ La même chose s'applique pour les parenthèses, les crochets ou les guillemets !

3.6.3. Appel d'une fonction

Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom (une fois de plus en respectant la casse) suivi d'une parenthèse ouverte (éventuellement des arguments) puis d'une parenthèse fermée :

```
Nom_De_La_Fonction();
```

Remarque :

✓ le point-virgule signifie la fin d'une instruction et permet au navigateur de distinguer les différents blocs d'instructions

✓ si jamais vous avez défini des arguments dans la déclaration de la fonction, il faudra veiller à les inclure lors de l'appel de la fonction (le même nombre d'arguments séparés par des virgules !)

```
Nom_De_La_Fonction(argument1, argument2);
```

3.6.4. Prototype d'une fonction

Le prototype d'une fonction est une description d'une fonction qui est définie plus loin dans le programme. On place donc le prototype en début de programme (avant la fonction principale *main()*).

Cette description permet au compilateur de « vérifier » la validité de la fonction à chaque fois qu'il la rencontre dans le programme, en lui indiquant :

- ✓ Le type de valeur renvoyée par la fonction
- ✓ Le nom de la fonction
- ✓ Les types d'arguments

Contrairement à la définition de la fonction, le prototype n'est pas suivi du corps de la fonction (contenant les instructions à exécuter), et ne comprend pas le nom des paramètres (seulement leur type).

Un prototype de fonction ressemble donc à ceci :

Type_de_donnee_renvoyee Nom_De_La_Fonction(type_argument1, type_argument2, ...);

Le prototype est une instruction, il est donc suivi d'un point-virgule !

Voici quelques exemples de prototypes :

void Affiche_car(char, int);

int Somme(int, int);

3.6.5. Les arguments d'une fonction

Il est possible de passer des arguments à une fonction, c'est-à-dire lui fournir une valeur ou le nom d'une variable afin que la fonction puisse effectuer des opérations sur ces arguments ou bien grâce à ces arguments.

Le passage d'arguments à une fonction se fait au moyen d'une liste d'arguments (séparés par des virgules) entre parenthèses suivant immédiatement le nom de la fonction. Le nombre et le type d'arguments dans la déclaration, le prototype et dans l'appel doit correspondre au risque, sinon, de générer une erreur lors de la compilation...

Un argument peut être :

- ✓ une constante
- ✓ une variable
- ✓ une expression
- ✓ une autre fonction

Avant la normalisation par l'ANSI, il était possible de faire une déclaration partielle d'une fonction, en spécifiant son type de retour, mais pas ses paramètres:

int f();

Cette déclaration ne dit rien sur les éventuels paramètres de la fonction f, sur leur nombre ou leur type, au contraire de :

int g(void); qui précise que la fonction g ne prend aucun argument.

3.6.6. Renvoie d'une valeur par une fonction

La fonction peut renvoyer une valeur (et donc se terminer) grâce au mot-clé *return*. Lorsque l'instruction *return* est rencontrée, la fonction évalue la valeur qui la suit, puis la renvoie au programme appelant (programme à partir duquel la fonction a été appelée). Une fonction peut contenir plusieurs instructions *return*, ce sera toutefois la première instruction *return* rencontrée qui provoquera la fin de la fonction et le renvoi de la valeur qui la suit.

La syntaxe de l'instruction return est simple :

return (valeur_ou_variable)

3.6.7. Exemples de fonction

- ✓ **Fonction inline**

Il s'agit d'une extension ISO C99, qui à l'origine vient du C++. Ce mot clé doit se placer avant le type de retour de la fonction. Il ne s'agit que d'une indication, le compilateur peut ne pas honorer la demande, notamment si la fonction est récursive. Dans une certaine mesure, les fonctionnalités proposées par ce mot clé sont déjà prises en charge par les instructions du préprocesseur. Beaucoup préfèreront passer par une macro, essentiellement pour des raisons de compatibilité avec d'anciens compilateurs ne supportant pas ce mot clé, et quand bien même l'utilisation de macro est souvent très délicat.

Le mot clé inline permet de s'affranchir des nombreux défauts des macros, et de réellement les utiliser comme une fonction normale, c'est à dire surtout sans effets de bord. À noter qu'il est préférable de classifier les fonctions inline de manière statique. Dans le cas contraire, la fonction sera aussi déclarée comme étant accessible de l'extérieur, et donc définie comme une fonction normale.

En la déclarant static inline, un bon compilateur devrait supprimer toute trace de la fonction et seulement la mettre *in extenso* aux endroits où elle est utilisée. Ceci permettrait à la limite de déclarer la fonction dans un fichier en-tête, bien qu'il s'agisse d'une pratique assez rare et donc à éviter. Exemple de déclaration d'une fonction inline statique :

```
staticinlineintmax(inta,int b)
{
        return(a > b)?a: b;
}
```

- ✓ **La fonction main**

Cette fonction est le point d'entrée du programme. La norme définit deux prototypes, qui sont donc portables:

```
int main(intargc,char*argv[]){/* ... */}
int main(void){/* ... */}
```

Le premier prototype est plus "général" : il permet de récupérer des paramètres au programme. Le deuxième existe pour des raisons de simplicité, quand on ne veut pas traiter ces arguments.

La fonction main prend deux paramètres qui permettent d'accéder aux paramètres passés au programme lors de son appel. Le premier, généralement appelé argc (*argument count*), est le nombre de paramètres qui ont été passés au programme. Le second, argv (*argument vector*), est la liste de ces paramètres. Les paramètres sont stockés sous forme de chaîne de caractères, argv est donc un tableau de chaînes de caractères, ou un pointeur sur un pointeur sur char. argc correspond au nombre d'éléments de ce tableau.

La première chaîne de caractères, dont l'adresse est dans argv[0], contient le nom du programme. Le premier paramètre est donc argv[1]. Le dernier élément du tableau, argv[argc], est un pointeur nul.

Valeur de retour

La fonction main retourne toujours une valeur de type entier. L'usage veut qu'on retourne 0 (ou EXIT_SUCCESS) si le programme s'est déroulé correctement, ou EXIT_FAILURE pour indiquer qu'il y a eu une erreur (Les macros EXIT_SUCCESS et EXIT_FAILURE étant définies dans l'en-tête <stdlib.h>). Il est possible par le programme appelant de récupérer ce code de retour, et de l'interpréter comme bon lui semble.

Voici un petit programme très simple qui affiche la liste des paramètres passés au programme lorsqu'il est appelé:

```
#include<stdio.h>

intmain(intargc,char*argv[])
{
        int i;
        for(i =0; i <argc; i++)
        printf("paramètre %i : %s\n",i,argv[i]);
        return0;
}
```

On effectue une boucle sur argv à l'aide de argc. Enregistrez-le sous le nom params.c puis compilez-le (cc params.c -o params).

- ✓ **Allocation dynamique**

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

Nous voulons lire 10 phrases au clavier et mémoriser les phrases en utilisant un tableau de pointeurs sur **char**. Nous déclarons ce tableau de pointeurs par:

```
char *TEXTE[10];
```

Pour les 10 pointeurs, nous avons besoin de 10*p octets. Ce nombre est connu dès le départ et les octets sont réservés automatiquement. Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...

La réservation de la mémoire pour les 10 phrases peut donc seulement se faire *pendant l'exécution du programme*. Nous parlons dans ce cas de l'***allocation dynamique*** de la mémoire.

- ✓ **La fonction malloc et l'opérateur sizeof**
  - **La fonction malloc**

La fonction **malloc** de la bibliothèque <*stdlib*> nous aide à localiser et à réserver de la mémoire au cours d'un programme. Elle nous donne accès au

tas *(heap)*; c'est-à-dire, à l'espace en mémoire laissé libre une fois mis en place le DOS, les gestionnaires, les programmes résidents, le programme lui-même et la pile *(stack)*.

malloc(<N>) : fournit l'adresse d'un bloc en mémoire de <N> octets libres ou la valeur zéro s'il n'y a pas assez de mémoire.

Sur notre système, le paramètre <N> est du type **unsigned int**. A l'aide de **malloc**, nous ne pouvons donc pas réserver plus de 65535 octets à la fois!

Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de 4000 caractères. Nous disposons d'un pointeur T sur **char** (**char *T**). Alors l'instruction:

T = malloc(4000) fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type. L'opérateur **sizeof** nous aide alors à préserver la portabilité du programme.

- ***L'opérateur unaire sizeof***

sizeof<var> : fournit la grandeur de la variable <var>
sizeof<const> : fournit la grandeur de la constante <const>
sizeof (<type>) : fournit la grandeur pour un objet du type <type>

Exemple

Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X est lue au clavier:

```
int X;
int *PNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

- ✓ ***exit***

S'il n'y a pas assez de mémoire pour effectuer une action avec succès, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande **exit** (de <*stdlib*>) et de renvoyer une valeur différente de zéro comme code d'erreur du programme (voir aussi chapitre 10.4).

- ✓ **La fonction free**

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque <*stdlib*>.

free(<Pointeur>) : libère le bloc de mémoire désigné par le <Pointeur>; n'a pas d'effet si le pointeur a la valeur zéro.

- La fonction **free** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par **malloc**.
- La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.

Si nous ne libérons pas explicitement la mémoire à l'aide free, alors elle est libérée automatiquement à la fin du programme.

CHAP. IV. STRUCTURES DES DONNEES

C'est un ensemble de données reliées logiquement, et dont l'organisation permet la manipulation individuelle ou collective de ces données.
Cette notion de structure des données est indépendante de tout langage de programmation. Un exemple élémentaire est le type entier (int).
Les actions possibles sont l'affectation, la lecture au clavier, les opérations +,*,-, /, …
Un exemple plus complexe, défini comme type construit dans les langages de programmation est le tableau (type []). Il contient divers éléments d'un même type de base. Les actions sont : définir/demander la taille, lire/écrire un élément par son numéro, …

4.1. STRUCTURE DES DONNEES

Dans le même esprit de l'indépendance par rapport à un langage de programmation, une structure des données se spécifie par la description logique du contenu et des actions possibles sur les données sans connaître sa programmation effective.

Une structure des données définit une abstraction des données et de leurs manipulations et cache l'implémentation.
La spécification logique est simplement celle de l'entier mathématique.
On décrit en général l'ensemble des actions possibles en quatre classes :

- ✓ Les constructeurs (fabriquer une donnée à partir de valeurs de base)
- ✓ Les sélecteurs (interroger la structure des données sur ces valeurs de base)
- ✓ Les modificateurs
- ✓ Les itérateurs (parcourir toutes les données de la structure des données)

Exemples d'actions pour le tableau :

- ✓ Modificateur : t[i]=a ;
- ✓ Sélecteur : if (t[i]==a) …
- ✓ Itérateur : index de l'élément [i]

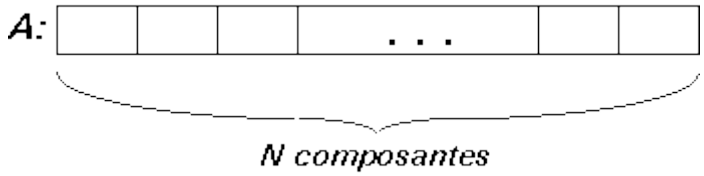
4.2. TABLEAU

L'une des variables structurées les plus populaires est appelée tableaux. Les tableaux sont disponibles dans tous les langages de programmation et servent

à résoudre une multitude de problèmes. Le traitement des tableaux en C ressemble celui des autres langages de programmation.

4.2.1. Les tableaux à une dimension

Un tableau A est dit unidimensionnel ou un vecteur s'il est formé d'un nombre entier N de variables simples du même type, qui sont appelées les *composantes* du tableau. Alors le nombre de composantes N est la *dimension* du tableau.



Exemple

int MOIS[12]={1,2,3,4,5,6,7,8,9,10,11,12};

Déclaration de tableaux

Type NomTableau[N];

En C, le nom d'un tableau est le représentant de l'adresse du premier élément du tableau. Les adresses des autres composantes sont calculées (automatiquement) relativement à cette adresse.

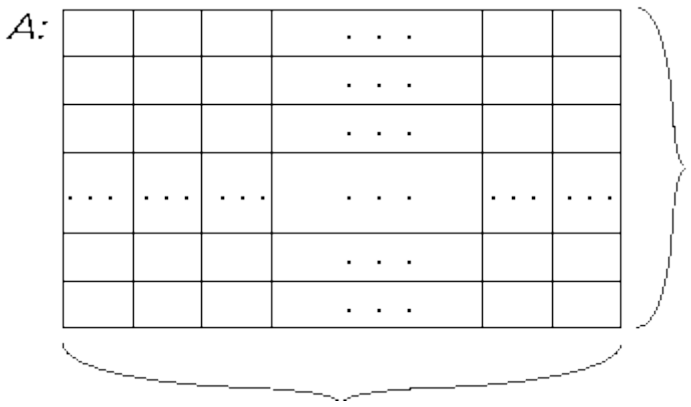
Exemple

```
short A[5] = {1200, 2300, 3400, 4500, 5600};
```

4.2.2. Les tableaux à deux dimensions

En C, un tableau à deux dimensions A est une matrice selon l'approche mathématique qu'on peut interpréter comme un tableau (unidimensionnel) de dimension N dont chaque composante est un tableau (unidimensionnel) de dimension M.

On appelle N le nombre de lignes du tableau et M le nombre de colonnes du tableau. N et M sont alors les deux dimensions du tableau. Un tableau à deux dimensions contient donc N*M composantes.



Déclaration d'un tableau à deux dimensions en C

Type NomTabl[N][M];

Accès à un tableau à deux dimensions en C

NomTableau[N][M] ;

4.3. CHAINE DE CARACTERES

Il n'existe pas de type spécial *chaîne* ou *string* en C. Une chaîne de caractères est traitée comme un *tableau à une dimension de caractères* (vecteur de caractères). Il existe quand même des notations particulières et une bonne quantité de fonctions spéciales pour le traitement de tableaux de caractères.

Char NomVariable [N] ;

Exemple

char NOM [20];

Lors de la déclaration, nous devons indiquer l'espace à réserver en mémoire pour le stockage de la chaîne.

La représentation interne d'une chaîne de caractères est terminée par le symbole **'\0'** (NUL). Ainsi, pour un texte de **n** caractères, nous devons prévoir **n+1** octets.

Malheureusement, le compilateur C ne contrôle pas si nous avons réservé un octet pour le symbole de fin de chaîne; l'erreur se fera seulement remarquer lors de l'exécution du programme ...

Le nom d'une chaîne est le représentant de *l'adresse du premier caractère* de la chaîne. Pour mémoriser une variable qui doit être capable de contenir un texte de N caractères, nous avons besoin de N+1 octets en mémoire:

char TXT[10] = "Felicitation !";

En général, les tableaux sont initialisés par l'indication de la liste des éléments du tableau entre accolades:

char CHAINE[] = {'F','e','l','l','y','\0'};

Pour le cas spécial des tableaux de caractères, nous pouvons utiliser une initialisation plus confortable en indiquant simplement une chaîne de caractère constante:

char CHAINE[] = "Hello";

Lors de l'initialisation par **[]**, l'ordinateur réserve automatiquement le nombre d'octets nécessaires pour la chaîne. Nous pouvons aussi indiquer explicitement le nombre d'octets à réserver, si celui-ci est supérieur ou égal à la longueur de la chaîne d'initialisation.

4.3.1. Accès aux éléments d'une chaîne

L'accès à un élément d'une chaîne de caractères peut se faire de la même façon que l'accès à un élément d'un tableau. En déclarant une chaîne par:

char A[6];
Nous avons défini un tableau A avec six éléments, auxquels on peut accéder par:

A[0], A[1], ... , A[5]

4.3.2. Les fonctions de <stdio.h>

Comme nous l'avons déjà vu au chapitre 4, la bibliothèque <*stdio*> nous offre des fonctions qui effectuent l'entrée et la sortie des données. A côté des fonctions **printf** et **scanf** que nous connaissons déjà, nous y trouvons les deux fonctions **puts** et **gets,** spécialement conçues pour l'écriture et la lecture de chaînes de caractères.

- Affichage de chaînes de caractères

printf avec le spécificateur de format **%s** permet d'intégrer une chaîne de caractères dans une phrase.

En plus, le spécificateur **%s** permet l'indication de la largeur *minimale* du champ d'affichage. Dans ce champ, les données sont justifiées à droite. Si on indique une largeur minimale négative, la chaîne sera justifiée à gauche. Un nombre suivant un point indique la largeur *maximale* pour l'affichage.

puts est idéale pour écrire une chaîne constante ou le contenu d'une variable dans une ligne isolée.

Exemple

```
char TEXTE[] = "Voici une première ligne.";
puts(TEXTE);
```

- Lecture de chaînes de caractères

scanf avec le spécificateur **%s** permet de lire un mot isolé à l'intérieur d'une suite de données du même ou d'un autre type.

Exemple

```
char LIEU[25];
int JOUR, MOIS, ANNEE;
printf("Entrez lieu et date de naissance : \n");
scanf("%s %d %d %d", LIEU, &JOUR, &MOIS, &ANNEE);
```

Comme le nom d'une chaîne de caractères est le représentant de l'adresse du premier caractère de la chaîne, il ne doit pas être précédé de l'opérateur adresse '&' !

La fonction **scanf** avec plusieurs arguments présuppose que l'utilisateur connaisse exactement le nombre et l'ordre des données à introduire! Ainsi, l'utilisation de **scanf** pour la lecture de chaînes de caractères est seulement conseillée si on est forcé de lire un nombre fixé de mots en une fois.

gets est idéal pour lire une ou plusieurs lignes de texte (p.ex. des phrases) terminées par un retour à la ligne.

gets(<Chaîne>)

Exemple

```
char LIGNE[200];
gets(LIGNE);
```

4.3.3. Les fonctions de string

La bibliothèque <*string*> fournit une multitude de fonctions pratiques pour le traitement de chaînes de caractères. Voici une brève description des fonctions les plus fréquemment utilisées.

Dans le tableau suivant, <n> représente un nombre du type **int**. Les symboles <s> et <t> peuvent être remplacés par :

- ✓ une chaîne de caractères constante
- ✓ le nom d'une variable déclarée comme tableau de **char**
- ✓ un pointeur sur **char** (voir chapitre 9)

Fonctions pour le traitement de chaînes de caractères

- ✓ strlen (chaine) : fournit la longueur de la chaîne ***sans*** compter le '\0' final
- ✓ strcpy(chaine 1, chaine 2) : copie chaine 2 vers chaine 1
- ✓ strcat(chaine 1 , chaine 2) : ajoute chaine 2 à la fin de chaine 1
- ✓ strcmp(chaine 1, chaine 2) : compare les deux chaines lexico graphiquement et fournit le résultat (négatif, zéro ou positif).
- ✓ Strncpy(chaine 1, chaine 2, n) : copie au plus n caractères de chaine 2 vers la chaine 1.
- ✓ Strncat(chaine 1, chaine 2, n) : ajoute au plus n caractères de chaine 2 à la fin de la chaine 1.

Remarques

- ✓ Comme le nom d'une chaîne de caractères représente une adresse fixe en mémoire, on ne peut pas 'affecter' une autre chaîne au nom d'un tableau alors il faut bien copier la chaîne caractère par caractère ou utiliser la fonction **strcpy** respectivement **strncpy**:

```
strcpy(A, "Hello");
```

- La concaténation de chaînes de caractères en C ne se fait pas par le symbole '+' comme en langage algorithmique ou en Pascal. Il faut ou bien copier la deuxième chaîne caractère par caractère ou bien utiliser la fonction **strcat**ou **strncat**.
- La fonction **strcmp** est dépendante du code de caractères et peut fournir différents résultats sur différentes machines.

4.3.4.Les fonctions de stdlib

La conversion de nombres en chaînes de caractères et vice-versa.

Les trois fonctions définies ci-dessous correspondent au standard ANSI-C et sont portables. La variable chaine peut être remplacé par :

- une chaîne de caractères constante
- le nom d'une variable déclarée comme tableau de **char**
- un pointeur sur **char**

Conversion de chaînes de caractères en nombres

- atoi(chaine) : retourne la valeur numérique représente par chaine comme int
- atol(chaine) : retourne la valeur numérique représente par chaine comme long
- atof(chaine) : retourne la valeur numérique représente par chaine comme double

Règles générales pour la conversion:

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

4.3.5. Les fonctions de ctype

Les fonctions de <*ctype*> servent à classifier et à convertir des caractères. Les symboles nationaux (é, è, ä, ü, ß, ç, ...) ne sont pas considérés. Les fonctions de <*ctype*> sont indépendantes du code de caractères de la machine et favorisent la portabilité des programmes.

Fonctions de classification et de conversion

Les fonctions de classification suivantes fournissent un résultat du type **int** différent de zéro, si la condition respective est remplie, sinon zéro.

- Isupper(caractère) : si caractère est une majuscule.
- islower(caractère) : si caractère est une minuscule.
- Isdigit(caractère) : si caractère est un chiffre décimal
- Isalpha(caractère), isalnum(caractère), isxdigit(caractère), isspace(caractère).

Les fonctions de ***conversion*** suivantes fournissent une valeur du type **int** qui peut être représentée comme caractère; la valeur originale de caractère reste inchangée:

- tolower(chaine) : retourne la chaine convertie en minuscule
- toupper(chaine) : retourne la chaine convertie en majuscule.

4.4. POINTEUR

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de pointeurs, c.-à-d. à l'aide de variables auxquelles on peut attribuer les adresses *d'autres variables*.

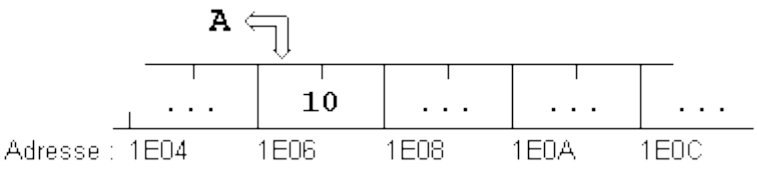
En C, les pointeurs jouent un rôle primordial dans la définition de fonctions ; Comme le passage des paramètres en C se fait toujours par la valeur, les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions. Ainsi le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs.

En outre, les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficients et fournissent souvent la seule solution raisonnable à un problème. Ainsi, la majorité des applications écrites en C profitent extensivement des pointeurs.

4.4.1. Adressage direct

Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéderdirectement à cette valeur.

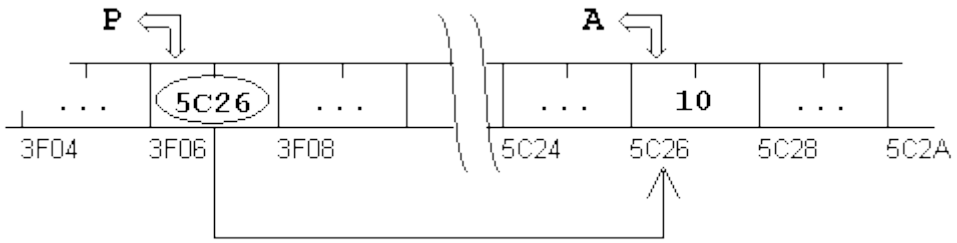
```
short A;
A = 10;
```



4.4.2. Adressage indirect

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée pointeur. Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

Remarque

Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence:

- ✓ Un pointeur est une variable qui peut 'pointer' sur différentes adresses.
- ✓ Le nom d'une variable reste toujours lié à la même adresse.

4.4.3. Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin

- ✓ d'un opérateur 'adresse de' **&** pour obtenir l'adresse d'une variable.

- ✓ d'un opérateur 'contenu de' ***** pour accéder au contenu d'une adresse.
- ✓ d'une syntaxe de déclarationpour pouvoir déclarer un pointeur.

&NomVariable : fournit l'adresse de la variable <NomVariable>

L'opérateur **&** nous est déjà familier par la fonction **scanf,** qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

L'opérateur **&** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

Soit P un pointeur non initialisé et A une variable (du même type) contenant la valeur 10.

Alors **P = &A** affecte l'adresse de la variable A à la variable P.

***NomPointeur :** désigne le contenu de l'adresse référencée par le pointeur NomPointeur

Exemple

Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé:

```
P = &A;
B = *P;
*P = 100;
```

P pointe sur A, le contenu de A (référencé par *P) est affecté à B, et le contenu de A (référencé par *P) est mis à 20.

Déclaration d'un pointeur

Type *NomPointeur

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données.

4.4.4. Les opérations élémentaires sur pointeurs

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

Priorité de * et &

- ✓ Les opérateurs ***** et **&** ont la même priorité que les autres opérateurs unaires (la négation **!**, l'incrémentation **++**, la décrémentation **--**). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.
- ✓ Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple

Après l'instruction

P = &X; et les deux écritures sont pareils A=X+1 et A=*P+1

Le pointeur NUL

Seule exception: La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.

```
int *P;
P = 0;
```

Finalement, les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soit P1 et P2 deux pointeurs sur **int**, alors l'affectation

P1 = P2; copie le contenu de P2 vers P1. P1 pointe alors sur le même objet que P2.

Après les instructions:

```
int A;
int *P;
P = &A;
```

A désigne le contenu de A, &A désigne l'adresse de A, P désigne l'adresse de A et *P désigne le contenu de A

En outre &P désigne l'adresse du pointeur P et *A est illégal puisque A n'est pas un pointeur.

4.4.5. Allocation dynamique de mémoire

Nous avons vu que l'utilisation de pointeurs nous permet de mémoriser économiquement des données de différentes grandeurs. Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de *l'allocation dynamique* de la mémoire.

Revoyons d'abord de quelle façon la mémoire a été réservée dans les programmes que nous avons écrits jusqu'ici.

Déclaration statique de données

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des données. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation. Nous parlons alors de la déclaration statique des variables.

Exemple

```
float A, B, C;        /* réservation de 12  octets */
short D[10][20];      /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                      /* réservation de 10  octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                      /* réservation de 40  octets */
```

Pointeurs

Le nombre d'octets à réserver pour un *pointeur* dépend de la machine et du 'modèle' de mémoire choisi, mais il est déjà connu lors de la compilation. Un pointeur est donc aussi déclaré statiquement. Supposons dans la suite qu'un pointeur ait besoin de p octets en mémoire. (En DOS: p =2 ou p = 4)

Exemple

```
double *G;      /* réservation de p    octets */
char *H;        /* réservation de p    octets */
float *I[10];   /* réservation de 10*p octets */
```

Chaînes de caractères constantes

L'espace pour les *chaînes de caractères constantes* qui sont affectées à des pointeurs ou utilisées pour initialiser des pointeurs sur **char** est aussi réservé automatiquement:

```
char *J = "Bonjour !";
        /* réservation de   p+10      octets */
float *K[] = {"un", "deux", "trois", "quatre"};
        /* réservation de 4*p+3+5+6+7 octets */
```

4.5. STRUCTURES

Il peut être intéressant quelquefois d'associer plusieurs caractéristiques à une variable. Ainsi, un point d'un plan 10 est défini par une abscisse et une Ordonnée. Associer ces deux caractéristiques pour définir un nouveau type peut être réalisé au moyen du mot-clef **struct.** On définit un tel nouveau type ainsi :

```
structpt {
float x;
float y;
};
```

Par la suite, une variable pourra êtredéclarée de ce type ainsi :

struct pt origine;

Et l'accès (lecture et écriture) aux différents champs se fera ainsi :

```
origine.x = 0; /* ecriture */
origine.y = 0; /* ecriture encore*/
origine.x+origine.y /* lecture */
```

Comme taper **struct pt** pour chaque nouveau point n'est pas agréable, on peut utiliser le mot-clef **typedef** ainsi (on suppose la structure pt déjà définie) :

Typedef struct pt point;

Et par la suite, déclarer ainsi les variables :
point origine;
L'habitude veut que ces deux étapes soient faites en même temps. Onécrira donc plutôt :

typedef structpt {
float x;
float y;
} ;

4.6. LES UNIONS

Une variable peut, dans certains cas, avoir plusieurs types différents, selon le cas dans lequel on est, sans pourtant avoir en même temps deux types différents. Une telle propriété est implantable grâce à la notion d'union :

union type u
{
int i;
float x;
}
type u var;

Après les déclarations suivantes, si l'on sait que var représente un entier.
Onaccédera à sa valeur par un var.i, alors que s'il représente un décimal, on utilisera var.x. Cependant, l'affectation d'une valeur d'un type donné écrase la valeur précédente de var, même si elle était d'un autre type. Qui plus est, il ne faut surtout pas croire que si l'on fait var.i = 3, alors var.x vaudra 3!

4.7. LES TYPES ENUMERES

Il s'agit d'une manière pratique de définir un ensemble ni de valeurs pour une variable, par exemple un ensemble d'états.

enum jour {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche} ;

CHAP. V. GESTION DES FICHIERS

Les entrées/sorties en langage C se font par l'intermédiaire d'entités logiques, appelés flux, qui représentent des objets externes au programme, appelés fichiers. Un fichier peut être ouvert en lecture, auquel cas il est censé nous fournir des données ou ouvert en écriture, auquel cas il est destiné à recevoir des données provenant du programme. Un fichier peut être à la fois ouvert en lecture et en écriture. Une fois qu'un fichier est ouvert, un flux lui est associé. Un flux d'entrée est un flux associé à un fichier ouvert en lecture et un flux de sortie un flux associé à un fichier ouvert en écriture. Tous les fichiers ouverts doivent être fermés avant la fin du programme.

Lorsque les données échangées entre le programme et le fichier sont de type texte, la nécessité de définir ce qu'on appelle une ligne est primordiale. En langage C, une ligne est une suite de caractères terminée par le caractère de fin de ligne (inclus) : '\n'. Par exemple, lorsqu'on effectue des saisies au clavier, une ligne correspond à une suite de caractères terminée par ENTREE. Puisque la touche ENTREE termine une ligne, le caractère généré par l'appui de cette touche est donc, en C standard, le caractère de fin de ligne soit '\n'. Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (*buffer*), ce qui permet de réduire le nombre d'accès aux périphériques (disque...).

Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture) ...Ces informations sont rassemblées dans une structure dont le type, FILE *, est défini dans stdio.h. Un objet de type FILE * est appelé *flot de données* (en anglais, stream).

Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande **fopen**. Cette fonction prend comme argument le nom du fichier, négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction **fclose**.

5.1. OUVERTURE ET FERMETURE D'UN FICHIER

5.1.1. La fonction fopen

fopen("*nom-de-fichier*","*mode*")

La valeur retournée par fopen est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur NULL. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction fopen est égale à NULL afin de détecter les erreurs (lecture d'un fichier inexistant...).

Le premier argument de fopen est le nom du fichier concerné, fourni sous forme d'une chaîne de caractères. On préférera définir le nom du fichier par une constante symbolique au moyen de la directive #define plutôt que d'expliciter le nom de fichier dans le corps du programme.

Le second argument, *mode*, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré. On distingue

- les *fichiers textes*, pour lesquels les caractères de contrôle (retour à la ligne ...) seront interprétés en tant que tels lors de la lecture et de l'écriture ;
- les *fichiers binaires*, pour lesquels les caractères de contrôle ne se sont pas interprétés.
- Les différents modes d'accès sont les suivants :

"r"	ouverture d'un fichier texte en lecture
"w"	ouverture d'un fichier texte en écriture
"a"	ouverture d'un fichier texte en écriture à la fin
"rb"	ouverture d'un fichier binaire en lecture
"wb"	ouverture d'un fichier binaire en écriture
"ab"	ouverture d'un fichier binaire en écriture à la fin

"r+"	ouverture d'un fichier texte en lecture/écriture
"w+"	ouverture d'un fichier texte en lecture/écriture
"a+"	ouverture d'un fichier texte en lecture/écriture à la fin
"r+b"	ouverture d'un fichier binaire en lecture/écriture
"w+b"	ouverture d'un fichier binaire en lecture/écriture
"a+b"	ouverture d'un fichier binaire en lecture/écriture à la fin

Ces modes d'accès ont pour particularités :

- Si le mode contient la lettre r, le fichier doit exister.
- Si le mode contient la lettre w, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu.
- Si le mode contient la lettre a, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Trois flots standard peuvent être utilisés en C sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- stdin (standard input) : unité d'entrée (par défaut, le clavier) ;
- stdout (standard output) : unité de sortie (par défaut, l'écran) ;
- stderr (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

5.1.2. La fonction fclose

Elle permet de fermer le flot qui a été associé à un fichier par la fonction fopen. Sa syntaxe est :

fclose(*flot*)où *flot* est le flot de type FILE* retourné par la fonction fopen correspondant.

La fonction fclose retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

5.2. LES ENTREES-SORTIES FORMATEES

5.2.1. La fonction d'écriture fprintf

La fonction fprintf, analogue à printf, permet d'écrire des données dans un fichier. Sa syntaxe estfprintf(*flot*,"*chaîne de contrôle*",*expression-1*, ..., *expression-n*)où *flot* est le flot de données retourné par la fonction fopen. Les spécifications de format utilisées pour la fonction fprintf sont les mêmes que pour printf.

5.2.2. La fonction de saisie fscanf

La fonction fscanf, analogue à scanf, permet de lire des données dans un fichier. Sa syntaxe est semblable à celle de scanf :

fscanf(*flot*,"*chaîne de contrôle*",*argument-1*,...,*argument-n*)où *flot* est le flot de données retourné par fopen. Les spécifications de format sont ici les mêmes que celles de la fonction scanf.

5.2.3. Impression et lecture de caractères

Les fonctions **fgetc** et **fputc** permettent respectivement de lire et d'écrire un caractère dans un fichier. La fonction fgetc, de type int, retourne le caractère lu dans le fichier. Elle retourne la constante EOF lorsqu'elle détecte la fin du fichier. Son prototype est int fgetc(FILE* *flot*);où *flot* est le flot de type FILE* retourné par la fonction fopen. Comme pour la fonction getchar, il est conseillé de déclarer de type int la variable destinée à recevoir la valeur de retour de fgetc pour pouvoir détecter correctement la fin de fichier.

La fonction fputc écrit *caractere* dans le flot de données :

Int fputc(int *caractere*, FILE **flot*)

Elle retourne l'entier correspondant au caractère lu (ou la constante EOF en cas d'erreur).
Il existe également deux versions optimisées des fonctions fgetc et fputc qui sont implémentées par des macros. Il s'agit respectivement de getc et putc. Leur syntaxe est similaire à celle de fgetc et fputc :

intgetc(FILE* *flot*);
intputc(int *caractere*, FILE **flot*)

5.3. RELECTURE D'UN CARACTERE

Il est possible de replacer un caractère dans un flot au moyen de la fonction ungetc :

Int ungetc(int *caractere*, FILE **flot*);

Cette fonction place le caractère *caractere* (converti en unsigned char) dans le flot *flot*. En particulier, si *caractere* est égal au dernier caractère lu dans le flot, elle annule le déplacement provoqué par la lecture précédente. Toutefois, ungetc peut être utilisée avec n'importe quel caractère (sauf EOF).

5.4. LES ENTREES-SORTIES BINAIRES

Les fonctions d'entrées-sorties binaires permettent de transférer des données dans un fichier sans transcodage. Elles sont donc plus efficaces que les fonctions d'entrée-sortie standard, mais les fichiers produits ne sont pas portables puisque le codage des données dépend des machines.

Elles sont notamment utiles pour manipuler des données de grande taille ou ayant un type composé. Leurs prototypes sont :

size_tfread(void **pointeur*, size_t *taille*, size_t *nombre*, FILE **flot*);

size_tfwrite(void **pointeur*, size_t *taille*, size_t *nombre*, FILE **flot*);

Où *pointeur* est l'adresse du début des données à transférer, *taille* la taille des objets à transférer, *nombre* leur nombre. Rappelons que le type size_t, défini dans stddef.h, correspond au type du résultat de l'évaluation de sizeof. Il s'agit du plus grand type entier non signé.

La fonction fread lit les données sur le flot *flot* et la fonction fwrite les écrit. Elles retournent toutes deux le nombre de données transférées.

5.5. POSITIONNEMENT DANS UN FICHIER

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en *mode séquentiel* : les données du fichier sont lues ou écrites les unes à la suite des autres. Il est également possible d'accéder à un fichier en *mode direct*, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier. La fonction fseek permet de se positionner à un endroit précis ; elle a pour prototype :

int fseek(FILE *_flot_, long _deplacement_, int _origine_);

La variable _deplacement_ détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine ; il est compté en nombre d'octets. La variable _origine_ peut prendre trois valeurs :

- SEEK_SET (égale à 0) : début du fichier ;
- SEEK_CUR (égale à 1) : position courante ;
- SEEK_END (égale à 2) : fin du fichier.

La fonction int rewind(FILE *_flot_) permet de se positionner au début du fichier. Elle est équivalente à fseek(_flot_, 0, SEEK_SET);
La fonction long ftell(FILE *_flot_) retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).

CHAP VI. LES BASES DE LA PROGRAMMATION EN C#

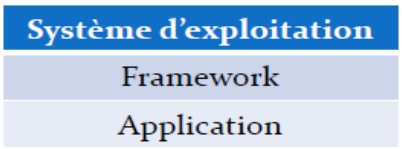
6.1. GENERALITES

La programmation est un ensemble de jeux d'instructions qui sera exécuté dans un ordinateur.

- ✓ Les étapes de programmation:
- ✓ Spécification de problème (cahier des charges)
- ✓ Analyse du problème
- ✓ Conception du problème
- ✓ Implémentation
- ✓ Et test

C#.NET est un langage orienté objet (LOO) ; Il est un produit Microsoft avec tant d'autres, tels que le VB, C++ etc.…

Le FRAMEWORK .net qui est une plate-forme reliant le langage au Système d'exploitation, une couche entre l'application et Windows.



Cette infrastructure offre un vaste accès :

- ✓ L'ensemble de système d'exploitation
- ✓ Une collection d'objets installables pour créer les programmes
- ✓ Des routines d'exécution des programmes.

Il existe d'autres concepts faisant objets de ce nouvel environnement de développement :

- ✓ CLR (Common Language Runtime) qui gère les exécutions de programme, le runtime peut être considéré comme un agent qui manage le code au moment d'exécution, qui l'exécute, fournit des services essentiels comme la gestion de la mémoire, la gestion des threads et l'accès distant.
- ✓ Assembly : pour les installations de programme, les mises à jour, l'utilisation des composants propres au programme ou partager avec

d'autres programmes, pour gérer les versions, éviter les problèmes des conflits des composants, on utilise les assembly.

- ✓ Pour utiliser son langage de programmation, on a besoin d'un éditeur (IDE=l'Integrated Development Environment : Environnement de développement intégré de Visual Studio 2010 de Microsoft. Il permet de dessiner l'interface (les fenêtres, les boutons, List, Image...) et d'écrire le code C#.

L'architecture .NET (nom choisi pour montrer l'importance accordée au réseau, amené à participer de plus en plus au fonctionnement des applications grâce aux services Web), technologie appelée à ses balbutiements NGWS (*Next Generation Web Services), consiste en une couche* Windows, ou plutôt une collection de DLL librement distribuable et maintenant directement incorporée dans le noyau des nouvelles versions de Windows.
Quelques caractéristiques de C#

- ✓ C# est un langage de haut niveau ;
- ✓ C# est souple c'est-à-dire il peut être exécuté localement ou à distance ;
- ✓ C# est multi cibles ;
- ✓ C# est prêt pour internet ;
- ✓ C# est purement orienté objet ;
- ✓ C# peut être utilisé pour créer, avec une facilité incomparable, des applications Windows et Web ;
- ✓ C# devient le langage de prédilection d'ASP.NET qui permet de créer des pages Web dynamiques avec programmation côté serveur.

Un peu d'histoire sur le langage C#

Le langage C# s'inscrit parfaitement dans la lignée C → C++ → C# .
Le langage C++ a ajouté les techniques de programmation orientée objet au langage C (mais la réutilisabilité promise par C++ ne l'a jamais été qu'au niveau source).
Le langage C# ajoute au C++ les techniques de construction de programmes sur base de composants prêts à l'emploi avec propriétés et événements, rendant ainsi le développement de programmes nettement plus aisé. La notion de briques logicielles aisément réutilisables devient réalité.

6.2. PROGRAMMATION ORIENTEE OBJET

Jusqu'à maintenant, l'activité de programmation a toujours suscité des réactions diverses allant jusqu'à la contradiction totale.

- ✓ Pour certains, il ne s'agit que d'un jeu de construction enfantin, dans lequel il suffit d'enchaîner des instructions élémentaires (en nombre restreint) pour parvenir à résoudre n'importe quel problème.
- ✓ Pour d'autres au contraire, il s'agit de produire (au sens industriel du terme) des logiciels avec des exigences de qualité qu'on tente de mesurer suivant certains critères.

Les qualités d'un logiciel :

- ✓ l'exactitude : est l'aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation ;
- ✓ la robustesse : est l'aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation ;
- ✓ l'extensibilité : est la facilité avec laquelle un programme pourra être adapté pour satisfaire à une évolution des spécifications ;
- ✓ la réutilisabilité : est la possibilité d'utiliser certaines parties (modules) du logiciel pour résoudre un autre problème ;
- ✓ la portabilité : est la facilité avec laquelle on peut exploiter un même logiciel dans différentes implémentations ;
- ✓ l'efficience : est le temps d'exécution, taille mémoire...

Les apports de la POO

- ✓ **Objet :** C'est là qu'intervient la P.O.O fondée justement sur le concept d'objet, à savoir une association des données et des procédures (qu'on appelle alors méthodes) agissant sur ces données.

Méthodes + Données = Objet

- ✓ **Encapsulation :** Cela signifie qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par l'intermédiaire de ses méthodes, qui jouent ainsi le rôle d'interface obligatoire. On traduit parfois cela en disant que l'appel d'une méthode est en fait l'envoi d'un "message" à l'objet.

- ✓ **Classe :** Une classe n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes. Les objets apparaissent alors comme des variables d'un tel type classe (en P.O.O., on dit aussi qu'un objet est une "instance" de sa classe).

- ✓ **Héritage :** Il permet de définir une nouvelle classe à partir d'une classe existante (qu'on réutilise en bloc !), à laquelle on ajoute de nouvelles données et de nouvelles méthodes.

- ✓ **Polymorphisme :** Généralement, en P.O.O, une classe dérivée peut "redéfinir" (c'est-à-dire modifier) certaines des méthodes héritées de sa classe de base. Cette possibilité est la clé de ce que l'on nomme le polymorphisme, c'est-à-dire la possibilité de traiter de la même manière des objets de types différents, pour peu qu'ils soient tous de classes dérivées de la même classe de base. Plus précisément, on utilise chaque objet comme s'il était de cette classe de base, mais son comportement effectif dépend de sa classe effective (dérivée de cette classe de base), en particulier de la manière dont ses propres méthodes ont été redéfinies.

Structure d'un programme c#

```
// importation des classes

public class nomClass {

    //déclaration des variables
    …
    //déclaration des méthodes
    …
    }
```

Une classe décrit la structure des données du système ainsi que les opérations permettant de la manipuler. Une classe représente la description abstraite d'un ensemble d'objets possédant les mêmes caractéristiques. On peut parler également de type. La classe est le concept fondamental de toute technologie objet.

Exemples : la classe Voiture, la classe Personne.

Syntaxe :

```
public class nom_de_la_classe {
    .......
    }
```

Exemple : public class Personne {

```
    ......
    }
```

Un objet est une entité aux frontières bien définies, possédant une identité et encapsulant un état et un comportement. Un objet est une instance (ou occurrence) d'une classe.

Exemple : Félicien Masakuna est une instance de la classe Personne.

Instanciation d'un objet :

```
nom_de_la_classenom_de_l_objet;
nom_de_l_objet=new nom_de_la_classe([parametres]);
```

On a déclaré un objet avec la première ligne et on l'a instancié avec la deuxième ligne.

Exemple : Personne unetudiant=new Personne();

Un attribut représente un type d'information contenu dans une classe.

Exemples : vitesse courante, cylindrée, numéro d'immatriculation, etc. sont des attributs de la classe Voiture.

Exemple : public class Voiture {

```
    privateint vitesse;
    private string matricule;
    private string modele;
    }
```

Les attributs sont des variables en C#. Leur type est soit un type primitif (int, etc.), soit une classe fournie par la plate-forme (string, etc.).

La visibilité des attributs(Encapsulation)

Il existe trois types d'encapsulation qu'un une classe ou un attribut soit vu ou non à l'extérieur de sa définition (classe ou package)

- Public : un attribut défini comme public est vu dans toutes les autres classes qui contiennent un objet de cette classe.
- Protected : un attribut défini comme protected n'est vu que par les classes filles de cette classe qui portera le nom de la classe mère.
- private : un attribut défini comme private n'est pas dans d'autres classes.

Les attributs sont aussi appelés champs statiques en C#.

Attributs de classe

Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe (Attribut d'instance). Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un attribut de classe (static en C#) qui garde une valeur unique et partagée par toutes les instances de la classe. Les instances ont accès à cet attribut mais n'en possèdent pas une copie.
Un attribut de classe n'est donc pas une propriété d'une instance mais une propriété de la classe et l'accès à cet attribut ne nécessite pas l'existence d'une instance.
Les méthodes statiques et non statiques doivent respecter les règles suivantes.
Une méthode ou une fonction statique n'a accès qu'aux champs statiques de sa classe (toute référence à a dans g serait sanctionnée par le compilateur).

Une méthode statique peut appeler une autre méthode statique (de sa classe ou d'une autre) mais ne peut pas appeler une méthode non statique.

Une méthode non statique d'une classe (par exemple f dans l'exemple précédent) a accès à la fois aux champs statiques et aux champs non statiques de sa classe. Ainsi, f peut modifier a et b.

Une opération représente un élément de comportement (un service) contenu dans une classe. Nous ajouterons plutôt les opérations en conception objet, car cela fait partie des choix d'attribution des responsabilités aux objets.

Nous avons généralement cinq types de méthodes :

- ✓ Accesseurs (Getter),
- ✓ Modificateurs (Setter),
- ✓ Constructeurs,

- ✓ Destructeurs
- ✓ et la méthode principale.

Les accesseurs sont des méthodes qui renvoient les valeurs:

```
public class Personne{
string nom;
stringprenom;
public string getNom() {
return nom;
}
public string getPrenom() {
returnprenom;
}
}
```

Les accesseurs contiennent toujours un type de valeur de retour et optionnellement des paramètres.

Les modificateurs changent les valeurs des certains attributs, sa valeur de retour est toujours nul (void) et peut posséder des paramètres

On peut surcharger les méthodes, c'est à dire avoir plusieurs de même nom mais des paramètres différents

```
public class Personne {
string nom;
stringprenom;
int age;
public void setNom(string nom,stringprenom) {
this.nom = nom;
this.prenom = prenom;
}
public void setAge(int age) {
this.age = age;
} }
```

Le constructeur est la méthode qui initialise une instance.
Le constructeur porte le même nom que la classe et n'a pas de valeur de retour, même **void**doit être omis dans la définition de cette fonction et il doit être qualifié public

Une classe peut avoir tant de constructeurs et qui se différencient par le nombre et type de paramètres (Surcharge).

```
public class Personne {
publicPersonne(string nom, string prenom) {
this.nom = nom;
this.prenom = prenom;
}
publicPersonne(int age) {
this.age = age;
}
public Personne() { }

}
```

Parce que le constructeur construit un objet, le destructeur le détruit afin de libérer l'espace mémoire. Un destructeur porte le nom de la classe préfixé de ~ (caractère tilde) et n'accepte aucun argument :

```
public class Personne {
publicPersonne(string nom, string prenom) {
this.nom = nom;
this.prenom = prenom;
}
public~Personne() {
Console.writeLine("Destruction en cours de l'objet");
}
}
```

La méthode principale est la méthode qui permet d'exécuter sa classe. Un projet a généralement beaucoup de classes mais une seule classe est principale. Cette classe principale doit obligatoirement contenir une méthode

principale(main)

```
public static void main(String[] args)
    {…….}
```

Les relations

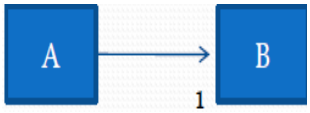
Une association représente une relation sémantique durable entre deux classes.

Exemple : une personne peut posséder des voitures.

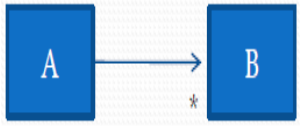
La relation *possède* est une association entre les classes Personne et Voiture.

Exemple de quelques relations et leurs implémentations

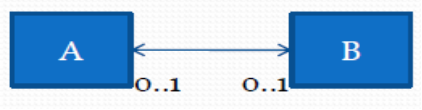
```
Public class A{
private B unB;
……
}
public class B{
……
    }
```




```
Public class A{
private B desB[];
……
}
public class B{
……
}
```




```
Public class A{
private B unB;
……
}
public class B{
private A unA;
……
}
```



Une agrégation est un cas particulier d'association non symétrique exprimant une relation de contenance.

- ✓ Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « contient », « est composé de ».
- ✓ Une composition est une agrégation plus forte impliquant que :
- ✓ Un élément ne peut appartenir qu'à un seul agrégat composite (agrégation non partagée) ;
- ✓ La destruction de l'agrégat composite entraîne la destruction de tous ses éléments (le composite est responsable du cycle de vie des parties).

```
public class Voiture {
private String modele;
private Moteur moteur;
private class Moteur {
private int puissance;
}
…
}
```

Une superclasse est une classe plus générale reliée à une ou plusieurs autres classes plus spécialisées (sous-classes) par une relation de généralisation.

- ✓ Les sous-classes « héritent » des propriétés de leur superclasse et peuvent comporter des propriétés spécifiques supplémentaires.

Exemple : les voitures, les bateaux et les avions sont des moyens de transport.

Ils possèdent tous une marque, un modèle, une vitesse, etc. et par contre, seuls les bateaux ont un tirant d'eau et seuls les avions ont une altitude…

```
public class Adherent : Personne {
private int iD;
}
public class Personne {
private string nom;
private string prenom;
}
```

Exemple sur l'héritage avec comme classe parent Personne et classe fille Etudiant :

```
public class Personne {
```

```
private string nom, postnom,;
public Personne(string nom, string postnom) {
this.nom = nom;
this.postnom = postnom;
}
}
```

```
public class Etudiant : Personne {
public Etudiant(string nom, String postnom, string matriucle) : base (nom, postnom) {
this.matriucle = matriucle;
}
}
```

Le qualificatif sealed

Le qualificatif **sealed** peut être appliqué à une classe pour empêcher toute dérivation à partir de cette classe.

```
sealed class Français : Pers
{
.....
}
```

Il devient maintenant impossible de créer une classe dérivée de Français.

```
publicEtudiant(string nom, string postnom, string prenom) : base(nom, postnom, prenom) {
}
stringmatriucle;
intnumeroPromotion;

public static void main(string[] args) {
newEtudiant(null, null, null);
}
}
```

Une classe abstraite est simplement une classe qui ne s'instancie pas directement mais qui représente une pure abstraction afin de factoriser des propriétés communes.

C'est le cas de *Moyen de Transport* dans l'exemple précédent à la page 73.

Le Package est le mécanisme général de regroupement d'éléments en POO

```
namespace humain;
abstract public class Personne {
….
}
```

Le framework regroupe les classes dans des espaces de noms. Il s'agit là d'une manière de partitionner les différentes (et parfois nombreuses) classes d'un programme et d'y voir ainsi plus clair dans l'organisation du programme. Par défaut, le nom donné à cet espace de noms est celui du projet (pour la classe Program d'un programme en mode console ou la classe dérivée de Form pour un programme Windows).

On peut retrouver un même espace de noms dans différents fichiers d'une même application (c'est d'ailleurs le cas par défaut si vous ajoutez une classe avec Visual Studio) :

```
namespace X
{
class Program { ..... }
}
namespace Y
{
class C { ..... }
}
```

Nous pouvons utiliser une classe qui a été développée par un quelconque monsieur, avant toute chose, nous devons l'importer.

Comment : using nom_complet_de_la_classe

Emplacement : avant la définition de la classe

Les packages existants :

- ✓ *System* pour accès aux types de base, accès à la console : Int32, Int64, Int16, Byte, Char, String, Float, Double, Decimal, Console et Type
- ✓ *System.Collections* pour les Collections d'objets : ArrayList, Hashtable, Queue, Stack, SortedList
- ✓ *System.IO* pour accès aux fichiers. File, Directory, Stream, FileStream, BinaryReader, BinaryWriter TextReader, TextWriter

- ✓ *System.Data.Common* pour accès ADO.NET aux bases de données : DbConnection, DbCommand, DataSet
- ✓ *System.Net* pour accès au réseau : Sockets,TcpClient, TcpListener UdpClient
- ✓ *System.Reflection* pour accès aux métadonnées : FieldInfo, MemberInfo, ParameterInfo
- ✓ *System.Security* pour contrôle de la sécurité : Permissions, Policy Cryptography
- ✓ *System.WinForms* pour composants orientés Windows : Form, Button, ListBox, MainMenu, StatusBar, DataGrid
- ✓ *System.Web.UI.WebControls* pour composants orientés Windows : Button, ListBox, HyperLink, DataGrid

NB : Mais il ya d'autres librairies qui ne sont pas gérées par le Framework qu'on doit ajouter comme étant des bibliothèques avant bien sûr leurs utilisations.

6.3. LECTURE ET ECRITURE DEPUIS LE CLAVIER

Contrairement à C, C# écrit avec l'instruction Console.WriteLine() ou Console.Write() et lit avec l'instruction Console.ReadLine() ou Console.Read().

6.4. STRUCTURES DE CONTROLE

Les langages C et C# partagent presque les mêmes structures de contrôle et avec les mêmes syntaxes. Il y a une des structures ajoutée au langage C# pour faciliter la manipulation des tableaux ou collections (foreach).

Foreach

L'instruction foreach permet de parcourir un tableau ainsi qu'une collection.

```
int[] ti = new int[]{10, 5, 15, 0, 20};

foreach (int i in ti)
Console.WriteLine(i);
```

Pour balayer un tableau de chaînes de caractères, on écrit (en déclarant et initialisant le tableau d'une autre manière) :

```
string[] noms = {"Kafunda", "Katalay", "Pierre"};
foreach (string nom in noms)
```

```
Console.WriteLine(nom);
```

L'instruction foreach en dernière ligne doit être lue : « pour chaque cellule du tableau noms (cellule de type string et que nous décidons d'appeler nom pour la durée du foreach), il faut exécuter l'instruction Console.WriteLine ».
Plusieurs instructions peuvent être spécifiées : il faut alors les entourer d'accolades, comme pour un if, for ou while.
foreach permet de balayer un tableau ou une collection mais l'opération est limitée à la lecture.
foreach ne permet pas de modifier le tableau ou la collection, vous devez pour cela utiliser une autre forme de boucle et recourir aux [] pour indexer la collection.
Pour afficher les arguments de la ligne de commande lors de l'exécution du programme, il suffit d'écrire :

```
staticvoidMain(string[] args)
{
foreach (string s in args) Console.WriteLine(s);
}
```

Les instructions break et continue peuvent être utilisées dans un foreach.

6.5. COLLECTIONS

Les collections sont des classes qui permettent d'implémenter des tableaux dynamiques, des piles, des listes chaînées éventuellement triées, etc.
Un objet conteneur contient donc des objets, organisés d'une manière ou d'une autre.
Les conteneurs d'objets contiennent des objets. Les collections peuvent contenir n'importe quel type. Si un transtypage ou un casting n'est pas nécessaire lors de l'introduction d'un élément dans le conteneur, il l'est lors du retrait. Avec les génériques, on spécifie le type des articles du conteneur, ce qui évite ces transtypages.

- ✓ **Les tableaux dynamiques**

La taille d'un tableau dynamique est automatiquement ajustée, si nécessaire, lorsque des éléments y sont ajoutés, ce qui n'est pas le cas pour les tableaux traditionnels. La classe ArrayList ou tableau dynamique implémente un « tableau dynamique d'objets ». Comme il n'y a pas de miracle en informatique, cette facilité a un coût (il faut bien des instructions en plus pour effectuer ces réorganisations quand cela s'avère nécessaire).

```
class Program
{
staticvoid Main ()
{
ArrayList liste = new ArrayList();
Pers p = new Pers {Nom="Jordan", Age=25}; al.Add(p);
liste.Add(new Pers{Nom="Gabin", Age=40});
for (int i=0; i< liste.Count; i++)
{
p = (Pers) liste[i];
p.Affiche();
}
}
}
```

Hormis les boucles for et foreach qui servent au balayage du tableau dynamique, il existe une autre technique qu'on nomme itérateur :

```
IEnumerator enu = liste.GetEnumerator();
while (enu.MoveNext())
Console.WriteLine(enu.Current);
```

MoveNext (à appliquer à l'objet itérateur) renvoie true quand l'objet suivant (par exemple un objetPers) est disponible. Dans ce cas, on trouve cet objet dans **enu.Current**.
Comme **enu.Current**, officiellement de type Object, est en réalité un objet Pers, c'est un objet Pers qui est affiché (ToString étant une méthode virtuelle, c'est la méthode ToString de Pers qui est appelée).
Généralement, il faudra copier **enu.Current** dans un objet de la classe Pers :

```
Pers p = (Pers)enu.Current;
```

- ✓ **La classe Stack**

Comparativement au tableau dynamique, l'accès à une pile est plus limité ; Un objet est toujours déposé au sommet de la pile et seul peut être retiré l'objet qui se trouve au sommet, selon le principe d'une pile d'assiettes.
La classe Stack implémente une pile. L'accès à une pile se fait selon le principe du LIFO (Last In, first Out en anglais).

Exemple 1

```
Stack entier = new Stack();
entier.Push(20);
entier.Push(40);
```

Exemple 2

```
Stack pile = new Stack();
pile.Push(new Pers{Nom="Felicien", Age=25});
pile.Push(new Pers{Nom="Pierre", Age=40});
for (int i=pile.Count; i>0; i--)
{
Pers p = (Pers)pile.Pop();
Console.WriteLine(p);
}
```

- ✓ **La classe Queue**

Un autre type de collection est appelé la classe Queue qui ressemble à une pile, sauf que l'accès est de type FIFO (first in, first out) ; Les éléments sont insérés d'un côté et retirés de l'autre.
Les insertions dans la file se font par Enqueue et les retraits par Dequeue. Peek permet d'inspecter l'élément de tête sans le retirer de la file.

- ✓ **Les listes triées**

La classe SortedList implémente une liste triée selon une clé et sans possibilité de doublons. Il faut distinguer la clé, par exemple l'identificateur d'une personne et la « valeur » associée à cette clé. La clé peut être de tout type et ainsi que les valeurs c'est-à-dire des objets.

```
SortedListsl = new SortedList();
Pers p = new Pers{ID=3333, Nom="Gaston", Age=27}; sl.Add(p.ID, p);

for (int i=0; i<sl.Count; i++)
{
Pers p = (Pers)sl[i];
}
for (int i=0; i<sl.Count; i++)
{
Pers p = (Pers)sl.GetByIndex(i);
Console.WriteLine(p);
}
```

Retrouver le nombre d'objets et retrouver une valeur via une position dans l'index.

```
foreach (Pers per in sl.Values)
Console.WriteLine(per);
Balayage de la collection des valeurs.
IDictionaryEnumerator
de = sl.GetEnumerator();
while (de.MoveNext())
{
Pers per = (Pers)de.Value;
Console.WriteLine(per);
}

foreach (object clé in sl.Keys)
Console.WriteLine(sl[clé]);
```

Balayage de la collection de clés et accès aux valeurs via la clé.

```
foreach (Pers per in sl.GetValueList())
Console.WriteLine(per);
```

Accès à la liste des valeurs renvoyée par GetValueList.
Cette forme est semblable à sl.Values.

```
foreach (object clé in sl.GetKeyList())
Console.WriteLine(sl[clé]);
```

- ✓ **La classe Hashtable**

La classe Hashtable implémente un conteneur parfois appelé « panier » (bag en anglais) ou dictionnaire. On y insère des objets sans ordre particulier, objets que l'on pourra retrouver plus tard sur la base d'une clé qui doit être unique. Les objets sont rangés dans différents compartiments sur la base de la clé. L'accès à un compartiment est rapide sur la base de la clé, mais un compartiment peut contenir plusieurs objets.

```
usingSystem.Collections;
Hashtableht = new Hashtable();
Pers p = new Pers{ID=7777, Nom="Gaston", Age=25};
ht.Add(p.ID, p);
p = (Pers)ht[7777];
if (p != null) Console.WriteLine(p.Nom);
elseConsole.WriteLine("Pas trouvé");
```

Il existe bien d'autres types de collections tels que StringDictionary, NameValueCollection…

6.6. TRY … CATCH… FINALLY

Le traitement des erreurs est en C# via le concept ou l'instruction Try catch. Il faut placer dans un bloc les différentes instructions susceptibles de poser problème.
Un deuxième bloc sert au traitement de l'erreur et un dernier bloc qui contiennent des instructions qui sont toujours exécutées, quel que soit le cheminement du programme.

```
try
{
//Codes susceptibles de provoquer une erreur
}
catch (Exception)
{
//Traitement de l'erreur
}
Finally  {  // Codes toujours exécutés}
```

6.7. CONVERSION DES TYPES

Faisons le point sur les conversions automatiques et présentons la notion de transtypage (*casting* en anglais). Peu de changement par rapport au C/C++ si ce n'est que C# se montre plus tatillon.
Dans tous les cas où une perte d'information est possible (par exemple une perte de précision), une conversion doit être explicitement spécifiée à l'aide de l'opérateur de *casting*. Quand une assignation est possible sans la moindre perte d'information, une conversion est automatiquement réalisée si nécessaire. On parle alors de conversion implicite.

Considérons les instructions suivantes pour illustrer le sujet (plusieurs de ces instructions sont marquées comme erreurs lors de la compilation) :

```
short s=2000;
int i = 100000;
i = s;
```

Deux mille dans un short : pas de problème.
Cent mille dans un int : pas de problème.

Aucun problème pour copier s dans i (toutes les valeurs possibles d'un short figurent dans les limites d'un int).

float f;
f = i + s;
Aucun problème : i + s est de type int (le short ayant été automatiquement promu en un int le temps du calcul) et les valeurs entières sont représentées avec exactitude y compris dans un float. Pas de problème de valeurs limites non plus.

s = i; Erreur car toute valeur d'un int ne peut pas être copiée dans un short. Peu importe le contenu de i au moment d'exécuter le programme (l'erreur est d'ailleurs signalée lors de la compilation).

s = (short) i; On évite certes l'erreur de syntaxe mais une valeur erronée est copiée dans s si i contient une valeur hors des limites d'un short.

La directive checked permet néanmoins de signaler l'erreur en cours d'exécution de programme.

float f = 2.1f; Ne pas oublier le suffixe f (sinon, erreur de syntaxe).
int i = s + f; Erreur de syntaxe car un réel (le résultat intermédiaire s+f est de type float) peut dépasser les limites d'un int (le type de la variable i qui reçoit ce résultat intermédiaire).
i = (int)(s + f); Le *casting* résout le problème (sans oublier qu'il fait perdre la partie décimale de tout réel, 2.1 et 2.99 devenant 2 tandis que -2.1 et -2.99 deviennent -2).

Si les *castings* permettent de forcer des conversions qui ne sont pas automatiquement réalisées par le compilateur, sachez que le compilateur n'admet pas n'importe quel casting. En gros, disons que C# refuse les conversions qui n'ont pas de sens : par exemple pour passer d'une chaîne de caractères à un réel. Pour passer d'une représentation sous forme d'une chaîne de caractères à une représentation sous forme d'entier ou de réel, vous devez utiliser la méthode *Parse* des classes Int32, Int64, Single ou Double.

6.8. FICHIERS

Le langage C# manipule les fichiers de façon très différente que le langage C, vu l'apparition de la programmation orientée objet qui est supportée par l'un et non par l'autre.

Nous ne saurons pas tout expliciter mais à titre informatif, regardons un peu cette différence par ce qui suit.

La manipulation des fichiers est possible à travers l'espace de noms **System.IO** qui contient un certain nombre de classes permettant d'effectuer les opérations sur les fichiers. Voici les catégories de classes permettant la manipulation des fichiers :

- ✓ la classe **DriveInfo** : ce classe contient les possibilités travers sa définition pour fournir des informations sur une unité de disque ;
- ✓ les classes **Directory** et **DirectoryInfo :** ces deux classes permettent la manipulation des répertoires (création d'un sous-répertoire, connaissance des répertoires et fichiers d'un répertoire donné, etc.) ;
- ✓ les classes **File** et **FileInfo** : ces deux classes fournissent des informations sur un fichier et permettent diverses manipulations (suppression, changement de nom, copie, etc.) mais sans permettre de lire ou d'écrire des fiches de ce fichier;
- ✓ les classes **Stream** et apparentées qui traitent les flots de données et permettent de lire et d'écrire dans le fichier.

Les détails sur chacun des classes seront étudiés l'année prochaine.

Illustrations

```
using System.IO;
string chemin =
System.Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
string[] lines = System.IO.File.ReadAllLines(@"" + chemin + "\\parametres.txt");
string[] parametres = new string[lines.Length];

for (int i = 0; i <lines.Length; i++){
string[] ligne = lines[i].Split(new char[] { ':' });
parametres[i] = ligne[1].Trim();
            }
```

Pour renommer un fichier (Anc.jpg qui doit devenir Nouv.jpg), on écrit (sans vérifier que Anc.jpg existe bien et que Nouv.jpg n'existe pas déjà) :

```
using System.IO;
```

```
.....
File.Move("Anc.jpg", "Nouv.jpg");
```

Pour que l'opération puisse être effectuée sans problème, il faut que Anc.jpg existe et que Nouv.jpg n'existe pas. Tout cela est traité ici :

```
try
{
File.Move("Anc.jpg", "Nouv.jpg");
}
catch (FileNotFoundException)
{
Console.WriteLine("Anc.jpg n'existe pas !");
}
catch (IOException)
{
Console.WriteLine("Nouv.jpg existe déjà !");
}
```

Généralement, on se contente d'intercepter Exception (qui reprend n'importe laquelle des deux exceptions précédentes) et de communiquer à l'utilisateur le contenu du champ Message de l'objet exc automatiquement créé lors du déclenchement de l'exception :

```
try {File.Move("Anc.jpg", "Nouv.jpg");}
catch (Exception exc) {Console.WriteLine(exc.Message);}
```

Pour copier un fichier, mais tout en évitant d'écraser un fichier existant qui porterait le même nom, on écrit :

```
try
{
File.Copy("Anc.jpg", "Nouv.jpg", false);
}
catch (FileNotFoundException)
{
Console.WriteLine("Anc.jpg n'existe pas !");
}
catch (IOException)
{
Console.WriteLine("Nouv.jpg existe déjà !");
}
```

Pour vérifier si un fichier existe :

```
if (File.Exists("Fich.dat")) .....
```

Pour vérifier si un fichier est en lecture seule (ne pas oublier que plusieurs attributs pourraient être à « vrai ») :

```
FileAttributes fa = File.GetAttributes("Fich.dat");
if ((fa & FileAttributes.ReadOnly)>0) .....
```

Pour cacher un fichier et le rendre en lecture seule :

```
FileAttributes fa = FileAttributes.ReadOnly | FileAttributes.Hidden;
File.SetAttributes("Fich.dat", fa);
```

Pour modifier la date de création d'un fichier :

```
File.SetCreationDate("Fich.dat", new DateTime(1789, 7, 14));
```

6.9. CONTROLES WINDOWS

Une fenêtre ne présente évidemment guère d'intérêt si elle reste vide. En général, elle contient des boutons de commande, des cases à cocher, des boîtes de liste, etc. Tous ces composants sont appelés « contrôles ».

Pour créer un « contrôle », par exemple un bouton de commande, il faut (toujours avec les méthodes ancestrales) traiter le message WM_CREATE adressé à la fenêtre. C'est en envoyant ce message que Windows signale à la fenêtre qu'elle doit s'initialiser (autrement dit : créer ses composants).

Pour créer le bouton, le programme appelle la fonction CreateWindow de l'API Windows (un bouton n'est en effet rien d'autre qu'une fenêtre particulière) sans se tromper dans les onze arguments de la fonction. Souvent, un seul argument erroné suffit à provoquer le « plantage » du programme. Rien ne doit en effet être négligé dans les informations à connaître et rien ou presque n'est effectué par défaut (il est toujours question ici des techniques ancestrales). Rien non plus évidemment de visuel ou d'intuitif avec ces techniques.

Pour agir sur un composant (par exemple pour modifier le libellé d'un bouton de commande), il faut connaître le message à envoyer à ce composant ainsi

que les informations complémentaires à ce message. Connaître tout cela demandait un investissement considérable.
C# et l'environnement .NET vont rendre les choses à la fois bien plus conviviales et aisées.
Mais il reste cependant utile de comprendre, au moins dans les grandes lignes, les principes de base de la programmation Windows sans l'aide des outils modernes.
Il existe deux d'applications à developper avec C# ; les interfaces avec windows forms et celles avec WPF (Windows Presentation Foundation).

WPF est la spécification graphique de Microsoft .NET 3.0. Il intègre le langage descriptif XAML qui permet de l'utiliser d'une manière proche d'une page HTML pour les développeurs.

WPF est préinstallé avec Vista. Il est aussi possible de l'installer sur Windows XP (à partir du Service Pack 2) et Windows Server 2003.

Il est une surcouche logicielle à DirectX pour la fabrication d'interfaces utilisateurs en dehors d'applications ludiques, il remplace en fait Windows Forms (USER et GDI) hérité de Windows 1.0 et est entièrement vectoriel, pour le dessin comme pour le texte. Cela permet d'augmenter la taille des objets en fonction de la résolution de l'écran sans effet de pixellisation, il optimise également fortement la virtualisation des applications par prise en main à distance par la réduction des informations à faire transiter sur le réseau.

L'affichage du texte se fait au moyen des procédés ClearType, TrueType ou OpenType qui améliorent le lissage des caractères. Il supporte l'affichage de nombreux formats d'images ou vidéo comme MPEG, AVI, et bien sûr WMV de Microsoft.

WPF ne sert pas uniquement à afficher l'interface graphique des logiciels tels que traitement de texte, jeux, etc., mais il fournit également un environnement d'exécution évolué des pages web nommées Silverlight. Les applications Web, nommées XBAP (Xaml Browser Application, XAML Browser Applications), sont des programmes qui tournent dans Internet Explorer ou FireFox, sous Windows ou Mac OS (sous GNU/Linux la plate-forme s'appelle MoonLight). Par défaut, ces applications n'ont pas accès au système de fichiers pour la sécurité des données et du système d'exploitation, mais un "*manifest*" peut être installé pour témoigner de la confiance dans une application donnée. Ceci permet, par exemple, de faciliter le déploiement d'un logiciel sur un large nombre de machines.

Il y a séparation entre les données et leur présentation, les deux aspects étant traités par **WPF**. WPF gère les bases de données pour les applications ou le web, et il fournit des modèles de présentation.

WPF fournit tous les éléments d'interface graphique : "widgets", fenêtres, boutons, champs de texte, menus, listes, etc. La description de l'interface se fait en XAML bien qu'il soit toujours possible de générer des interfaces dynamiquement en code managé.

WPF fournit aux développeurs différents moyens de créer leurs propres composants, par agrégation (UserControl) ou dérivation (CustomControl) de composants existants.

L'une des particularités de WPF est de dissocier le contrôle, au sens "composant" du terme (Entrées/Sorties, Événements, etc.) de son graphisme. De fait, pour un contrôle donné, créer ou remplacer le graphisme (au sens large du terme, c'est-à-dire en incluant les animations, les sons, etc.) se fait de manière particulièrement aisée. On parle alors de "Template" de Control.

De fait, l'arbre XAML des composants, souvent nommé arbre "logique", est doublé d'un arbre Visuel, déterminé à l'exécution, et prenant en compte les différents "Templates" des contrôles de l'arbre logique, telles que définies via le XAML ou référencés, dans le Code Behind.

Dans ce cours, seules les fenêtres faites avec Windows Form qui seront exploitées.

La fenêtre

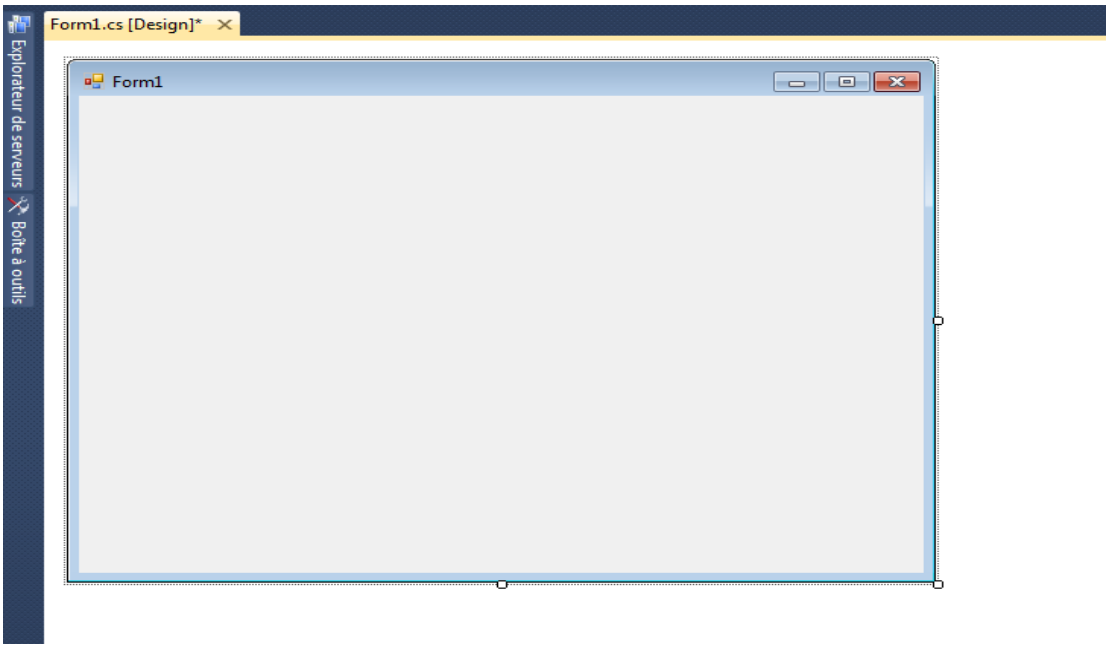
Les fenêtres (*windows* en anglais) jouent un rôle fondamental dans l'environnement Windows mais aussi dans n'importe quel environnement graphique. Au lieu de fenêtres, on parle aussi de formulaires, de formes ou encore de fiches (*forms* en anglais).
Pour l'utilisateur, une application se résume même souvent à une voire plusieurs fenêtres : une fenêtre connue généralement sous le nom de fenêtre principale de l'application ainsi que, éventuellement, une ou plusieurs fenêtres secondaires connues, selon les cas, sous les noms de fenêtres filles, de boîtes de dialogue ou encore de fenêtres enfants (dans le cas de fenêtres MDI, *Multiple Document Interface*, il s'agit de ces fenêtres limitées à leur fenêtres mères et qui peuvent être réarrangées en mosaïque ou en cascade).

Pour le programmeur, une application minimale consiste en :

- ✓ un objet « application » (qui n'a aucune matérialisation à l'écran et qui est automatiquement créé) contenant les fonctions de base (communes à tous les programmes) nécessaires au démarrage et au contrôle de l'application jusqu'à sa phase terminale (par exemple reconnaître la fin de l'application sur fermeture de la fenêtre principale) ;
- ✓ au moins un objet « fenêtre ».

```
public class LongTaskForm : Form
{

}
```

Une fenêtre est une classe que nous appelons Form ; pour créer une nouvelle fenêtre, on hérite la classe Form de l'espace des noms Windows.Forms.



Une fenêtre, comme les contrôles, changent de caractéristiques ou de présentations à travers leurs propriétés et relativement aux évènements.

Les Contrôles

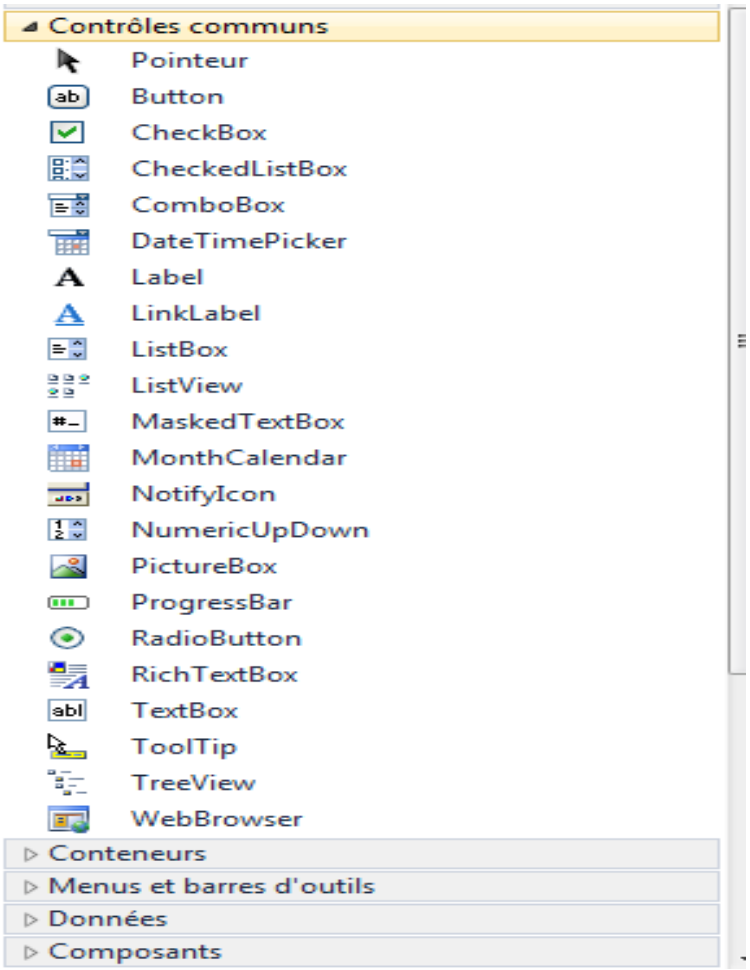
Un contrôle est un composant d'une fenêtre.
Il existe pas mal de contrôles constituants une fenêtre ; les contrôles sont catégorisés :

- ✓ conteneurs : les contrôles qui ont pour mission de contenir d'autres contrôles (panel, GroupBox, TabControle…).
- ✓ Menus et Barres d'outils : se servent pour les options de menus et barres d'outils (ContextMenuStrip, StatusStrip, MenuStrip…)

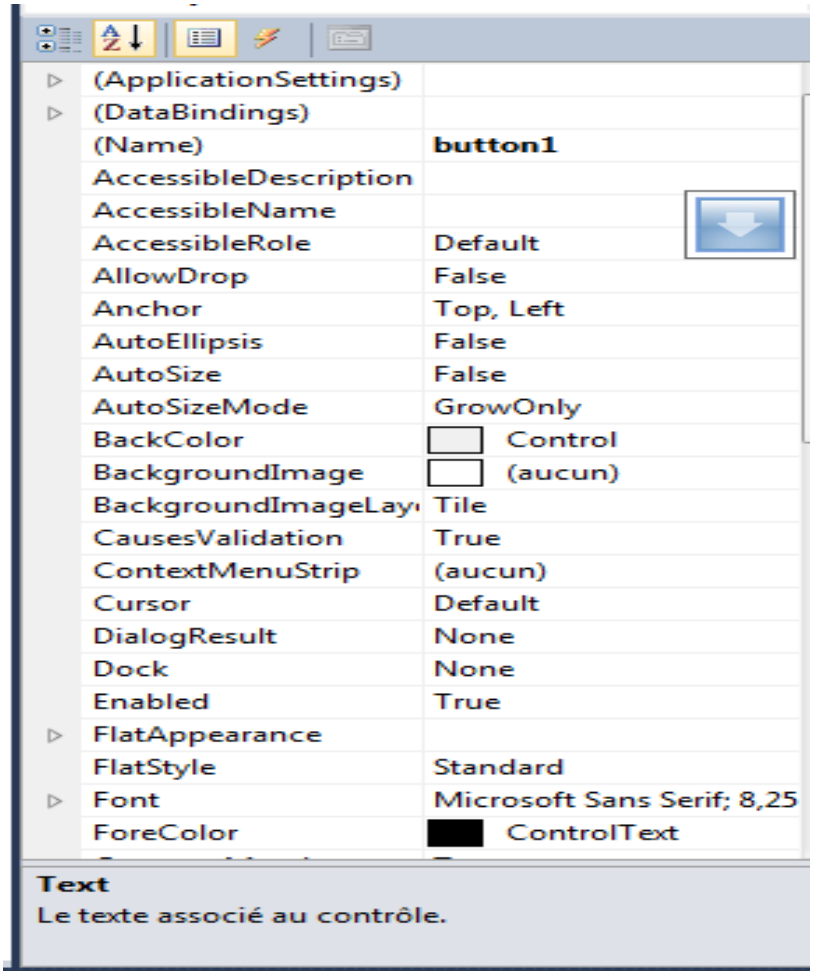
- ✓ Données : qui contiennent de contrôles permettant l'interaction avec les données (chat, datagridview, dataset…)
- ✓ Les boites de dialogue : qui contiennent des controles permettant l'interaction avec certaines ressources de la machine ou du système d'exploitation (ColorDialog, FontDialog, OpenFileDialog, SaveFileDialog,…)
- ✓ Etc.

Les contrôles contiennent des propriétés. Il existe les propriétés génériques c'est-à-dire utilisées par tous les contrôles tels que (Name, BackGround, Font, Text, …) et aussi des propriétés spécifiques à un contrôle tel que la propriété intervalle pour le contrôle Timer.

Voici la liste de quelques contrôles et ses détails seront exposés pendant les séances de travaux pratiques.



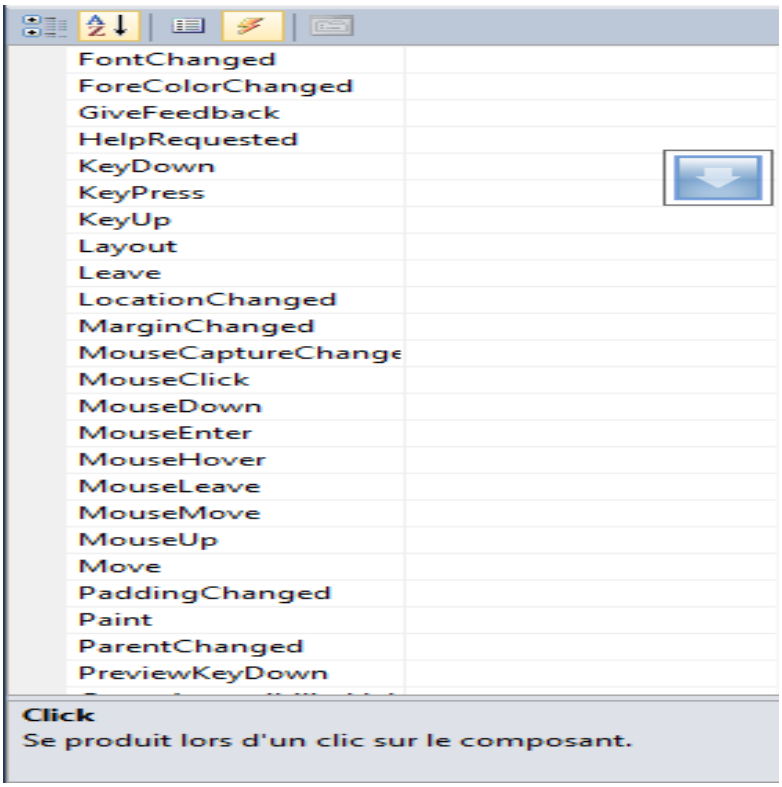
La liste de quelques propriétés :



Les contrôles gèrent aussi les évènements qui expriment les actions que l'utilisateur peut mener sur une fenêtre via les composants de cette dernière.

Nous avons les évènements tels que Key Press qui est déclenché après avoir enfoncé une touche, MouseEnter qui est déclenché après avoir cliqué sur la souris, FormLoad qui est déclenché au téléchargement du formulaire…

La liste de quelques évènements :



Syntaxes :

- ✓ Déclaration d'un contrôle

  private nomDeClasse nomObjet ;

```
private System.Windows.Forms.Label label1;
```

- ✓ Instanciation d'un contrôle

  nomObjet= new nomDeClasse() ;

```
this.label1 = new System.Windows.Forms.Label();
```

- ✓ Définition des propriétés

  nomObjet.propriete=valeur ;

```
            this.label1.AutoSize = true;
            this.label1.Font = new System.Drawing.Font("Segoe UI", 17F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.label1.ForeColor = System.Drawing.Color.Black;
            this.label1.Location = new System.Drawing.Point(17, 19);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(303, 31);
            this.label1.TabIndex = 58;
```

```
this.label1.Text = "Mise à jour du mot de passe";
```

✓ Ajout d'un évènement sur un contrôle

```
nomObjet.evenement += new System.EventHandler(methode);
```

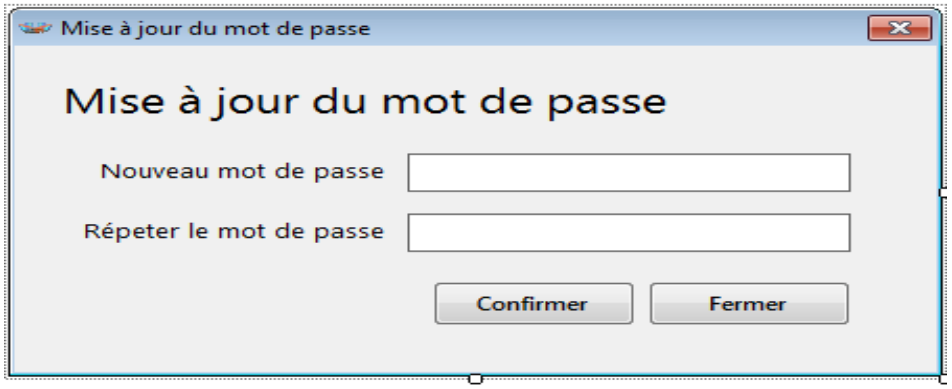
La méthode va contenir les codes qui seront réalisés lors de déclenchement de l'évènement.

```
this.label1.Click += new System.EventHandler(this.confirmerButton_Click);
```

✓ Ajout d'un contrôle sur le formulaire

```
this.Controls.Add(nomObjet);

this.Controls.Add(this.label2);
```

Exemple d'une fenêtre permettant la mise à jour d'un mot de passe



Cette fenêtre contient sept contrôles, trois labels ou étiquettes, deux textBox ou zones de texte et deux boutons de commande.

Pour créer un contrôle depuis le code, voici la procédure :

```
namespace ACGT_RH.ui
{
    partial class UpdatePassword
    {
       // declaration des objets
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label8;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.TextBox password2TextBox;
        private System.Windows.Forms.TextBox password1TextBox;
        private System.Windows.Forms.Button fermerButton;
        private System.Windows.Forms.Button confirmerButton;


        #region Windows Form Designer generated code
```

```
        private void InitializeComponent()
        {
              //instanciation des objets declarés

            System.ComponentModel.ComponentResourceManager resources = new
System.ComponentModel.ComponentResourceManager(typeof(UpdatePassword));
            this.label1 = new System.Windows.Forms.Label();
            this.label8 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.password2TextBox = new System.Windows.Forms.TextBox();
            this.password1TextBox = new System.Windows.Forms.TextBox();
            this.fermerButton = new System.Windows.Forms.Button();
            this.confirmerButton = new System.Windows.Forms.Button();
            this.SuspendLayout();

            // definitions de propriétés des objets
            // label1
            //
            this.label1.AutoSize = true;
            this.label1.Font = new System.Drawing.Font("Segoe UI", 17F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.label1.ForeColor = System.Drawing.Color.Black;
            this.label1.Location = new System.Drawing.Point(17, 19);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(303, 31);
            this.label1.TabIndex = 58;
            this.label1.Text = "Mise à jour du mot de passe";
            //
            // label8
            //
            this.label8.AutoSize = true;
            this.label8.Font = new System.Drawing.Font("Segoe UI", 9.75F);
            this.label8.ForeColor = System.Drawing.Color.Black;
            this.label8.Location = new System.Drawing.Point(30, 111);
            this.label8.Name = "label8";
            this.label8.Size = new System.Drawing.Size(152, 17);
            this.label8.TabIndex = 71;
            this.label8.Text = "Répeter le mot de passe";
            //
            // label2
            //
            this.label2.AutoSize = true;
            this.label2.Font = new System.Drawing.Font("Segoe UI", 9.75F);
            this.label2.ForeColor = System.Drawing.Color.Black;
            this.label2.Location = new System.Drawing.Point(38, 72);
            this.label2.Name = "label2";
            this.label2.Size = new System.Drawing.Size(144, 17);
            this.label2.TabIndex = 70;
            this.label2.Text = "Nouveau mot de passe";
            //
            // password2TextBox
            //
            this.password2TextBox.BorderStyle =
System.Windows.Forms.BorderStyle.FixedSingle;
            this.password2TextBox.Font = new System.Drawing.Font("Segoe UI", 10F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.password2TextBox.Location = new System.Drawing.Point(188, 109);
            this.password2TextBox.Name = "password2TextBox";
            this.password2TextBox.Size = new System.Drawing.Size(212, 25);
            this.password2TextBox.TabIndex = 69;
```

```
            this.password2TextBox.UseSystemPasswordChar = true;
            //
            // password1TextBox
            //
            this.password1TextBox.BorderStyle =
System.Windows.Forms.BorderStyle.FixedSingle;
            this.password1TextBox.Font = new System.Drawing.Font("Segoe UI", 10F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.password1TextBox.Location = new System.Drawing.Point(188, 70);
            this.password1TextBox.Name = "password1TextBox";
            this.password1TextBox.Size = new System.Drawing.Size(212, 25);
            this.password1TextBox.TabIndex = 68;
            this.password1TextBox.UseSystemPasswordChar = true;
            //
            // fermerButton
            //
            this.fermerButton.Font = new System.Drawing.Font("Segoe UI Semibold", 9F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.fermerButton.Location = new System.Drawing.Point(303, 153);
            this.fermerButton.Name = "fermerButton";
            this.fermerButton.Size = new System.Drawing.Size(97, 29);
            this.fermerButton.TabIndex = 67;
            this.fermerButton.Text = "Fermer";
            this.fermerButton.UseVisualStyleBackColor = true;
            this.fermerButton.Click += new
System.EventHandler(this.fermerButton_Click);
            //
            // confirmerButton
            //
            this.confirmerButton.Font = new System.Drawing.Font("Segoe UI Semibold",
9F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.confirmerButton.Location = new System.Drawing.Point(200, 153);
            this.confirmerButton.Name = "confirmerButton";
            this.confirmerButton.Size = new System.Drawing.Size(97, 29);
            this.confirmerButton.TabIndex = 66;
            this.confirmerButton.Text = "Confirmer";
            this.confirmerButton.UseVisualStyleBackColor = true;
            this.confirmerButton.Click += new
System.EventHandler(this.confirmerButton_Click);
            //
            // UpdatePassword
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(441, 212);

//ajout des controles sur la fenetre

            this.Controls.Add(this.label8);
            this.Controls.Add(this.label2);
            this.Controls.Add(this.password2TextBox);
            this.Controls.Add(this.password1TextBox);
            this.Controls.Add(this.fermerButton);
            this.Controls.Add(this.confirmerButton);
            this.Controls.Add(this.label1);
            this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedDialog;
            this.Icon = ((System.Drawing.Icon)(resources.GetObject("$this.Icon")));
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Name = "UpdatePassword";
            this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;
            this.Text = "Mise à jour du mot de passe";
```

```
            this.Load += new System.EventHandler(this.UpdatePassword_Load);
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion


    }
}
```

Mais dans la plupart des cas, une fenêtre et ses composants sont créés de façon graphique par le concepteur avec la méthode qu'on appelle cliquer-glisser et qui génère automatiquement des codes en background que le concepteur était censé taper pour avoir la dite fenêtre.
Cela est possible avec la zone boites à outils de l'EDI et la zone propriétés.

6.10. ACCES AUX BASES DES DONNEES

Dans ce point, nous allons nous intéresser aux techniques d'accès aux bases de données.

Pour illustrer le sujet, les exemples porteront essentiellement sur Access, un produit Microsoft particulièrement répandu. ADO.NET s'applique cependant à bien d'autres SGBD (Systèmes de gestion de bases de données) comme Oracle, MySql, SQL Serveur etc. Il suffit en général de changer la ligne qu'est la chaîne de connexion pour passer d'un SGBD à l'autre.
ADO.NET s'applique également aux données sous forme XML et aux fichiers Excel (mais sans évidemment le remplacer dans le traitement sur les données, qui peut être particulièrement élaboré).

Nous supposons que les principes généraux des bases de données sont connus, au moins dans les grandes lignes, que vous comprenez les opérations simples (SELECT, INSERT, UPDATE et DELETE) du langage SQL de manipulation de bases de données, et que vous savez utiliser, de manière élémentaire au moins, les logiciels Access et/ou SQL Server, ou encore la base de données que vous voulez ou devez utiliser.

À cause de son nom, on pourrait croire qu'ADO.NET est l'extension .NET du module d'accès aux bases de données ADO (ActiveX Data Objects), utilisé en

Visual Basic 6 (version précédant .NET, version qui peut être considérée comme obsolète, même si les programmes ainsi créés continuent à fonctionner). Malgré quelques points de ressemblance et quelques objets communs, ADO.NET est très différent et la connaissance d'ADO, sans être évidemment inutile, se révèle finalement d'un intérêt fort limité pour s'attaquer à ADO.NET.

ADO.NET est constitué d'un ensemble de classes qui agissent comme une interface entre votre programme et la base de données. Entre ADO.NET et la base de données, il peut exister encore toute une série de composants propres à la base de données, notamment des drivers :

- ✓ certains étant propres à un SGBD particulier (et optimisés pour celui-ci, par exemple SQL Server ou Oracle) ;
- ✓ d'autres étant plus généraux et s'appliquant à plusieurs bases de données (cas des drivers dits Ole-Db).

Cependant, tout cela reste heureusement transparent pour le programmeur utilisateur de la base de données.
Manipuler une base de données avec ADO.NET revient à manipuler quelques objets des classes formant ADO.NET. La connaissance du langage SQL est néanmoins toujours nécessaire.

Le premier de ces objets est l'objet de connexion. Comme son nom l'indique, il nous permet de nous connecter à une base de données (sur la machine locale ou sur une machine distante).

Les objets de connexion

Le premier des objets que l'on est amené à rencontrer est l'objet de connexion. Quel que soit le mode de travail (connecté ou déconnecté), il faut disposer, avant toute chose, d'un objet de connexion. C'est dans cet objet que nous allons spécifier les caractéristiques de la base de données à utiliser, notamment le nom et le type de base de données (Access, SQL Server, Oracle, etc.). Nous verrons que ces informations de connexion peuvent être spécifiées dans une chaîne de caractères et même dans un fichier de configuration, indépendant du programme. C'est aussi grâce à cet objet de connexion que nous pourrons, si nécessaire, découvrir par programme la description de la base de données (tables et champs de ces tables) et que nous obtiendrons les autres objets nécessaires à la programmation des bases de données.

Ces classes pour la connexion sont OleDbConnection, SqlConnection, OracleConnection et OdbcConnection.
Pour simplifier, nous les appellerons de manière générique *xyz*Connection, *xyz* étant à remplacer par OleDb, Sql, Oracle ou Odbc selon le type de base de données. Elles permettent d'établir la connexion entre le programme et la source de données (généralement une base de données locale ou distante).

La classe OleDbConnection est utilisée dans le cas de connexions fondées sur la technologie OleDb, c'est-à-dire dans le cas où les drivers utilisent des fonctions d'Ole-Db (il s'agit entre autres d'une implémentation assez générale de drivers de bases de données fournie par Microsoft).

ADO.NET utilise en interne OleDbConnection pour des connexions à des bases de données Access mais aussi aux systèmes de bases de données pour lesquels il n'y a pas (ou pas encore) de driver optimisé pour l'architecture .NET.
La classe SqlConnection est spécialement optimisée pour SQL Server (y compris SQL Server Express). La classe OleDbConnection peut certes être utilisée pour des accès à une base de données SQL Server, mais les performances sont alors sensiblement moindres. Les drivers utilisés quand on a recours à un objet SqlConnection court-circuitent Ole-Db et, du fait de leur optimisation, accroissent les performances d'un facteur deux.

La classe OracleConnection est optimisée pour la base de données Oracle, de la société du même nom.
Enfin, la classe OdbcConnection est destinée à ceux qui utilisent encore la technique ODBC (*Open Data Base Connectivity*) d'accès aux bases de données.
Les différentes classes *xyz*Connection présentent des propriétés et fonctions communes. Au nom près, on peut d'ailleurs, dans la pratique, considérer ces classes comme équivalentes. Nous n'utiliserons pas directement ces classes *xyz*Connection car nous programmerons de manière plus générique avec les fabriques de classes. De ce fait, nous utiliserons la classe (abstraite) DbConnection qui présente les mêmes propriétés, méthodes et événements.

Pour lire et/ou modifier une base de données, il faut d'abord créer, puis initialiser un objet de l'une de ces classes. Deux techniques sont possibles pour initialiser ces objets de connexion :

- ✓ spécifier une chaîne de connexion (celle-ci est constituée de texte qui reprend, en clair, les caractéristiques de la connexion) ;

- ✓ initialiser les propriétés de l'objet, comme on le fait pour tout autre objet.

Commençons par présenter ces classes *xyz*Connection. Nous présentons ici ces différentes classes même si, dans la pratique, nous créerons plutôt des objets de la classe DbConnection.

Cette dernière, de l'espace de noms System.Data.Common, est une classe abstraite qui reprend les méthodes et propriétés que doivent implémenter les classes *xyz*Connection.

Classes xyzConnection et classe générique DbConnection

*xyz*Connection ¨ DbConnection ¨ Component ¨ Object
using System.Data; // dans tous les cas
using System.Data.Common; // pour DbConnection
using System.Data.OleDb; // pour provider Ole-Db
using System.Data.Sql; // pour provider SQL Server
using System.Data.Oracle; // pour provider Oracle
using System.Data.Odbc; // pour provider ODBC

Constructeurs des classes xyzConnection

*xyz*Connection(); Crée un objet *xyz*Connection non initialisé.
*xyz*Connection(string chaînedeConnexion);

Crée un objet *xyz*Connection tout en spécifiant une chaîne de connexion.

L'objet reprend ainsi automatiquement les caractéristiques de la chaîne de connexion (le constructeur décortique la chaîne pour initialiser les propriétés de l'objet de connexion).

Principales propriétés des classes xyzConnection et DbConnection

ConnectionString str Chaîne de connexion. Il s'agit de la plus importante des propriétés car elle permet, en une seule chaîne, de remplacer toutes les autres propriétés.
ConnectionTimeout int Durée, en secondes, durant laquelle l'établissement d'une connexion (au serveur de bases de données) est tenté. Si au bout de ce laps de temps, la connexion n'est pas établie, elle est signalée en erreur. Par défaut, cette durée est de quinze secondes. Une valeur nulle indique une attente infinie, ce qu'il faut éviter : durant cette attente, le programme ne

réagit plus à aucune sollicitation et il n'y a aucun moyen de sortir de cette situation, sauf tuer le programme.

DataSource str Nom de la source de données (voir exemples plus loin). À utiliser dans le cas d'Access (et de manière générale des providers OleDb), sans oublier que l'on écrit Data Source, en deux mots, dans la chaîne de connexion.
PacketSize int Taille des paquets utilisés lors d'une communication avec SQL Server. Cette taille, qui est par défaut de 8 192 octets, peut varier entre 512 et 32 767. Augmenter la valeur par défaut peut améliorer les performances si vous transmettez des champs de texte de grande taille ou des images (avec contenus binaires directement dans la base de données).

Password str Mot de passe pour accéder à la base de données.
Provider str Identificateur du service (voir les chaînes de connexion).
ServerVersion Version du provider.
State État de la connexion. State peut prendre une ou plusieurs des valeurs de l'énumération
ConnectionState : Broken (connexion rompue), Closed, Connecting, Executing et Open.
WorkstationID str Nom de l'ordinateur à partir duquel on réclame la connexion.

Méthodes des classes xyzConnection et DbConnection

void Open(); Ouverture explicite de la connexion. Une exception est générée si l'ouverture ne peut être effectuée dans le délai imparti (par exemple parce que la base de données n'existe pas ou parce que vous n'avez pas les droits d'accès suffisants).

void Close(); Fermeture explicite de la connexion. Une fermeture implicite est toujours effectuée quand le programme se termine. Il est néanmoins préférable de fermer la connexion aussitôt qu'elle n'est plus nécessaire (voir ci-dessous l'optimisation réalisée par ADO.NET avec la *connection pool*).

Bien que nous n'encouragions pas du tout cette pratique (mais bien les fabriques de classes), voyons quand même comment ouvrir une connexion en utilisant l'une des classes *xyz*Connection (en supposant que connStr contienne la chaîne de connexion, ici, pour un accès à une base de données Access, d'où l'utilisation du provider Ole-Db) :

```
using System.Data;
```

```
using System.Data.OleDb;
.....
OleDbConnection oConn = new OleDbConnection(connStr);
try {oConn.Open();}

catch (Exception exc)
{
..... // message d'erreur dans exc.Message
}
```

Pour affiner le traitement d'erreur, on peut considérer que l'exception est plus précisément de type :
Notre programme, tel qu'il vient d'être écrit, ne serait applicable à un autre type de base de données (passer d'Access à SQL Server par exemple) qu'après de multiples changements (changer notamment tous les objets OleDb*xyz* en Sql*xyz*). Nous éviterons, grâce aux fabriques de classes, d'avoir à créer explicitement des objets *xyz*Connection propres à un type particulier de base de données.

La méthode Close ferme la connexion à la base de données, comme on ferme un fichier. Il ne faut cependant pas se méprendre en se fondant trop sur l'analogie avec des fichiers : la couche .NET du système d'exploitation garde en fait la connexion ouverte, dans ce que l'on appelle une connection pool. Cela permet d'accélérer considérablement une prochaine opération Open qui aurait des arguments déjà rencontrés (même provider, même base de données, mêmes caractéristiques d'utilisation, etc.). Les programmes sont en effet invités à garder les connexions ouvertes le moins longtemps possible afin de ne pas encombrer le serveur avec des connexions ouvertes mais peu ou pas utilisées. C'est particulièrement vrai en programmation ASP.NET où
de nombreux utilisateurs (aux commandes de leur navigateur) accèdent aux mêmes bases de données.

Les classes OleDbConnection, SqlConnection, OracleConnection et OdbcConnection héritent de la classe abstraite DbConnection. En vertu d'une des règles de la POO, les objets des quatre types précités sont considérés comme des objets de type DbConnection. Travailler avec des objets DbConnection présente l'avantage de pouvoir écrire des programmes indépendants du type de provider. C'est d'ailleurs cette technique qu'utilisent les fabriques de classes, avec le résultat que nous venons d'énoncer, à savoir l'indépendance vis-à-vis du type de base de données.

On parlera donc à l'avenir d'objets DbConnection plutôt que d'objets OleDbConnection, SqlConnection, OracleConnection ou OdbcConnection.
Avant de créer (enfin) un objet DbConnection, voyons comment spécifier les caractéristiques de la base de données dans ce que l'on appelle une chaîne de connexion.

Les chaînes de connexion

Les différentes informations contenues dans les propriétés des objets DbConnection peuvent être reprises dans ce que l'on appelle une « chaîne de connexion ». Il s'agit, sous forme de texte en clair, d'une chaîne de caractères au format bien particulier : "mot-clé=valeur; mot-clé=valeur" où plusieurs couples mot-clé/valeur peuvent être spécifiés. Cette chaîne de connexion, tout en étant plus simple à écrire, évite de devoir initialiser individuellement les différentes propriétés des objets de connexion. Rien n'empêche d'utiliser les deux techniques : chaîne de connexion complétée par des propriétés de l'objet de connexion.

Plutôt que de nous attarder sur la syntaxe des mots-clés et des valeurs, présentons des exemples de chaînes de connexion pour les bases de données Access et pour SQL Server.
Le site www.connectionstrings.com constitue une remarquable source d'informations, en donnant les chaînes de connexion pour un grand nombre de logiciels de base de données.

Cas d'une base de données Access

Pour une connexion à la base de données Biblio.mdb, créée sous Access et située dans le répertoire c:\Data de la machine locale, la chaîne de connexion est (utilisation ici des chaînes verbatim pour éviter de doubler les barres obliques car, par défaut, \ introduit un caractère de contrôle) :

```
@"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\Data\Biblio.mdb"
```

Avec les versions les plus récentes d'Access, la base de données est un fichier d'extension accdb et la chaîne de connexion est :

```
@"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=Biblio.accdb";
```

Au besoin, les différents éléments (par exemple c:\Data\Biblio.mdb) peuvent être délimités par ' (single quote).

Le nom de la base de données est spécifié dans la clé Data Source (attention à l'espace de séparation).
Si le fichier de base de données Access se trouve dans le répertoire courant de l'application (par défaut celui du fichier EXE de l'application), vous pouvez laisser tomber le nom du répertoire. Ce fichier pourrait se trouver sur une autre machine du réseau. Dans ce cas, spécifiez le chemin complet qui mène à cette autre machine (par exemple \\NomOrdi\Rep\Biblio.accdb), comme c'est le cas pour tout fichier distant mis en partage (le nom de la machine distante qui héberge la base de données étant préfixé de deux \).

Au moins le runtime d'Access doit avoir été installé sur la machine du client. Ce runtime est librement téléchargeable à partir du site de téléchargement de Microsoft (www.microsoft.com/downloads et effectuer une recherche sur Access runtime).
Ce qui est expliqué ici s'applique, dans les détails, à des programmes Windows. Dans le cas de programmes ASP.NET, il faut appeler la fonction Server.MapPath qui renvoie le nom complet du fichier quand on lui passe en argument un nom de fichier du répertoire (avec ASP.NET, le répertoire courant n'est pas celui de l'application).
Dans la chaîne de connexion, vous pouvez ajouter (User Id pouvant être abrégé en uid et Password en pwd) :

```
User Id=.....; Password=.....
```

Dans le cas d'une base de données Access protégée par un mot de passe, il faut ajouter :

```
Jet OLEDB:Database Password=.....;
```

Les autres attributs de la chaîne de connexion

Dans la chaîne de connexion, on peut également spécifier (voir les propriétés des classes *xyz*Connection pour la signification) :

Attributs des chaînes de connexion

- Timeout Avec Connect Timeout ou Connection Timeout comme synonymes.

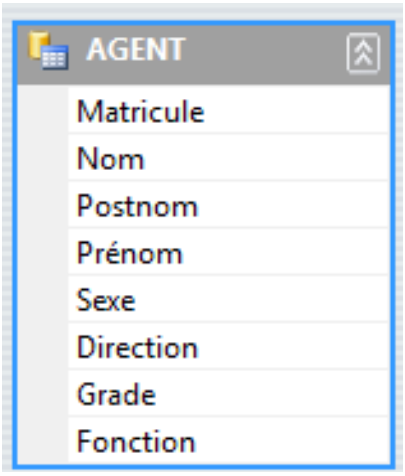
- Database Avec Initial Catalog comme synonyme (à n'utiliser que pour les connexions à SQL Server).
- Password Avec pwd comme synonyme.
- Server Avec Address, Addr et Network Address comme synonymes.
- Integrated Security Avec yes ou no comme valeurs possibles (ne fonctionne que pour l'objet SqlConnection).
- User ID Avec uid comme synonyme.
- Workstation ID Avec wsid comme synonyme.

6.11. ETAT DE SORTIE

Crystal Reports est un outil de génération des rapports ou des états à partir de différents sources de données.

Afin de rendre l'application ou la page contenant l'état indépendante des machines, il est préférable de créer une Datatable ou un modèle de données comme illustré ci-dessous ; et pendant la création d'un état de sortie il sera demandé de signaler le modèle à considérer.

C'est au niveau de codes qu'on doit signaler la chaine de connexion au rapport.



Modèle ou Datatable

Parmi les données d'entrées :

- ✓ Bases de données telles que Sybase, IBM DB2, Microsoft Access, Microsoft SQL Server, MySQL, Oracle …
- ✓ Classeurs Microsoft Excel
- ✓ Fichiers texte

- ✓ Fichiers HTML XML
- ✓ Toutes données accessibles par des liens ODBC, JDBC ou OLAP.

Traditionnellement Crystal Reports a été l'outil de reporting de choix fourni avec Visual Studio.

Nous pouvons avoir de rapports sous forme tabulaire ou sous forme graphique mais avec une multitude de modèles.

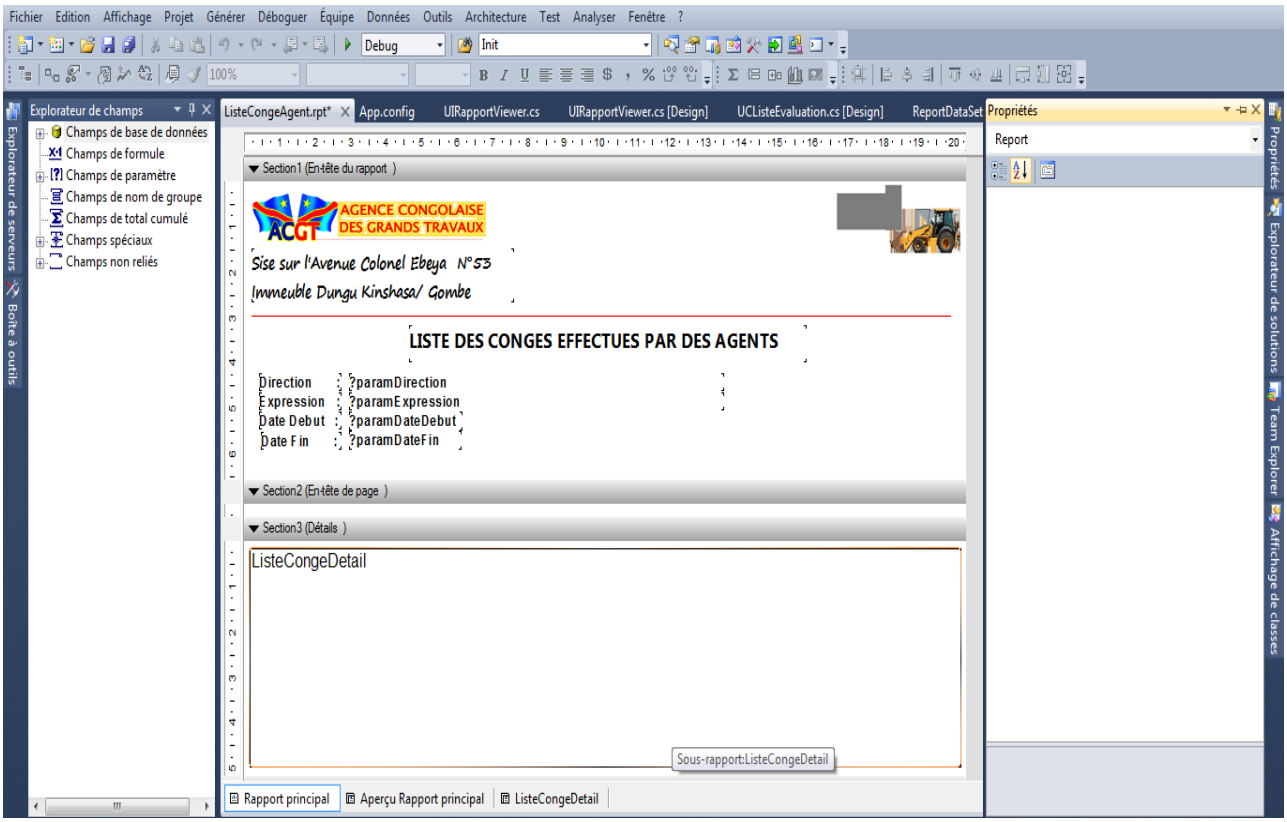
SAP fournit un nouveau control CrystalReportsViewer sans l'utilisation de WinForms pour créer une application Crystal Reports on a deux choix :

- ✓ soit on utilise directement le Template : Crystal Reports Application.
- ✓ sinon si on a déjà créé un projet, on ajoute le control CrystalReportsViewer dans notre page puis on ajoute les références :

    - o CrystalDecisions.CrystalReports.Engine
    - o CrystalDecisions.Shared

On peut permettre à l'utilisateur final certain contrôles de ce rapport tel que l'export de résultat vers différents formats (PDF, DOC, XLS …).

Il existe certaines informations utiles pour le papier mais qui ne sont pas gérées dans un fichier ou une base des données et ces informations sont envoyées au crystal sous forme des paramètres comme montré aux interfaces ci-dessous auxquelles on a renseigné le nom de la direction de l'agent, son grade, l'année d'évaluation depuis l'application.



La fenêtre de création de Crystal report contient une boite à outils car il s'agit aussi d'une application comme toutes les autres. La boite à outils Crystal contient par défaut quatre objets qui sont :

- ✓ Objet texte
- ✓ Objet ligne
- ✓ Objet cadre
- ✓ Et Pointeur

Crystal report contient une autre zone qu'on appelle **explorateur des champs** ; cet explorateur contient sept types ou catégories des champs qui étendent l'utilisateur de crystal. Les champs sont de fonctions prédéfinies allégeant l'utilisateur pendant la création de rapport.

Il s'agit de :

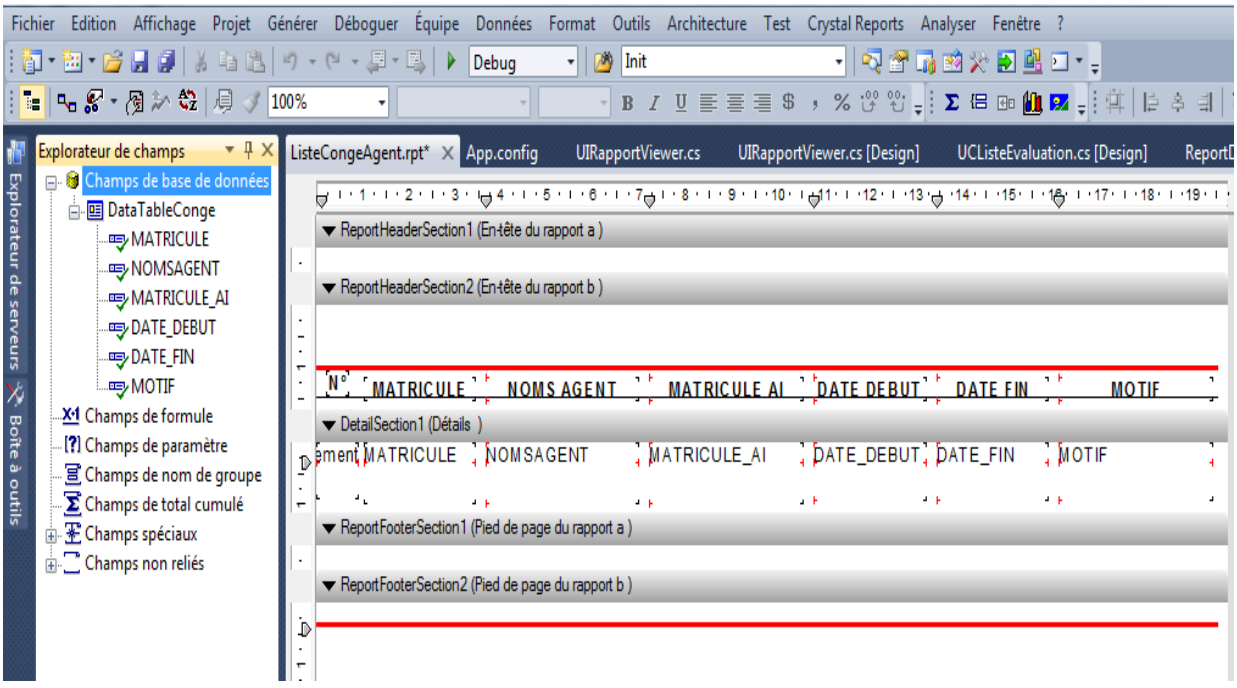
- ✓ Champs de base des données : à partir duquel on fournit les informations de sources de données ; pour notre cas, c'est le nom du modèle qui sera fournis.
- ✓ Champs de formule : on peut personnaliser une formule via les fonctions prédéfinies par Crystal report et cela est possible à travers ce deuxième type de champs.

- ✓ Les champs de paramètres : il existe certaines informations utiles pour le rapport mais générées à partir de l'application ou difficile voir même impossible de les avoir à partir d'une base des données ou d'un fichier. Ces informations sont appelées paramètres.
- ✓ Les champs spéciaux : les fonctions couramment utilisées telles que la date du jour, le numéro de la page, le nombre de pages etc.
- ✓ Les champs de nom de groupe, les champs non reliés et les champs de total cumulé.

Une page de crystal contient cinq zones appelées section pour sa conception :

- ✓ La première section contient l'entête du rapport ; la zone où seront renseignées les informations relatives au rapport dans son ensemble.
- ✓ La deuxième section contient l'entête de la page ; la zone où seront renseignées les informations relatives à la page.
- ✓ La troisième zone contient les détails de la page.
- ✓ La quatrième zone le pied de la page
- ✓ Et la cinquième contient le pied du rapport.

Exemple de détails du rapport avec Crystal report :



Les détails de ce concept seront vus pendant les travaux pratiques.

Exemple :

Les classes ou espaces de noms à importer :

```
using CrystalDecisions.CrystalReports.Engine;
using CrystalDecisions.Shared;
```

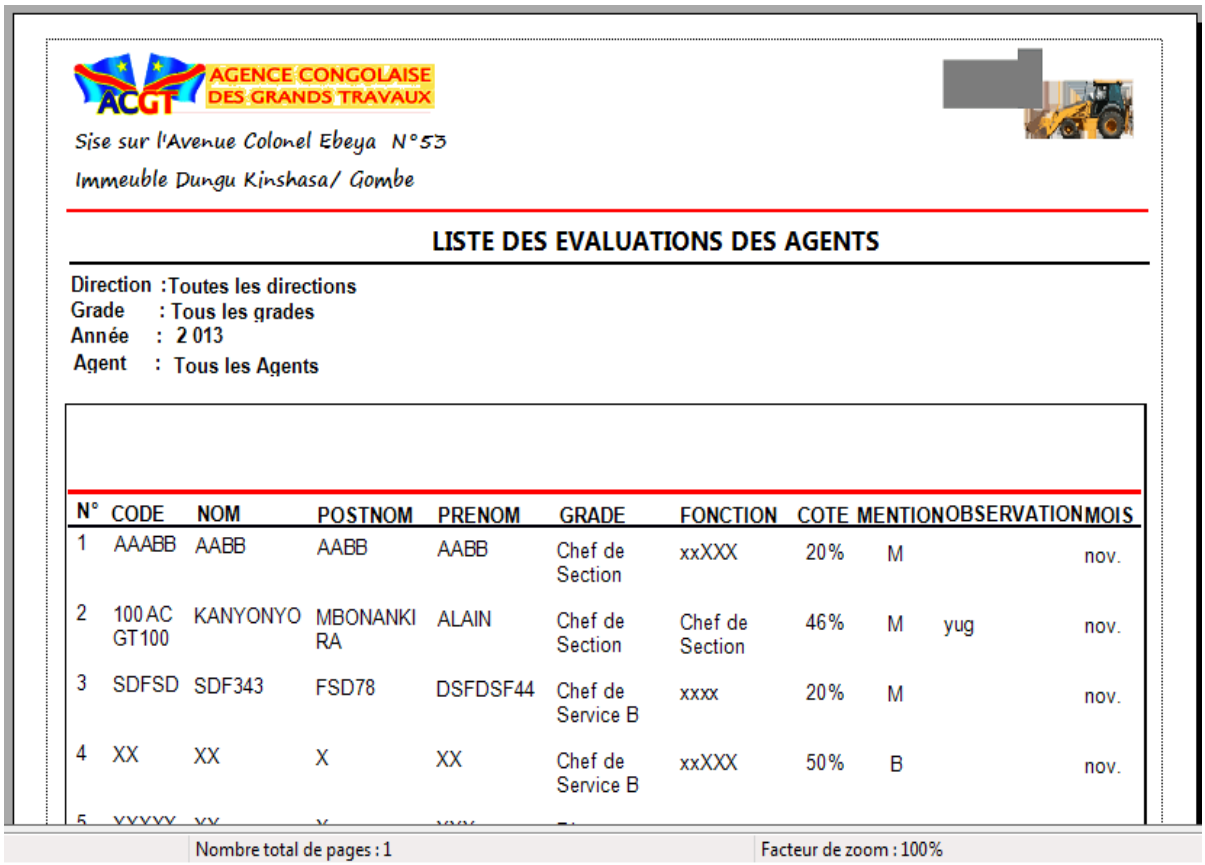
Les codes à lancer

```
DataTable dt =DAO.administration.SqlEvaluation.GetAgents(dir, grad, code,
anne,moi); // la méthode retourne une collection de type datatable
_ListeEvaluation.SetDataSource(dt);
```

Envoie des paramètres

```
_ListeEvaluation.SetParameterValue("paramDirection", direction);
_ListeEvaluation.SetParameterValue("paramGrade", grade);
_ListeEvaluation.SetParameterValue("paramAgent", codeAgent);
_ListeEvaluation.SetParameterValue("paramAnnee", annee);
this.crystalReportViewerListe.ReportSource = _ListeEvaluation;
```

Résultat de crystal report



AGENCE CONGOLAISE DES GRANDS TRAVAUX

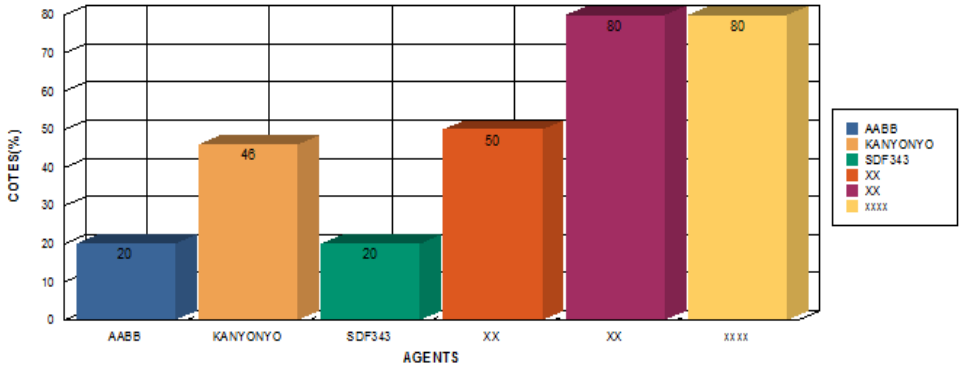
Sise sur l'Avenue Colonel Ebeya N°53

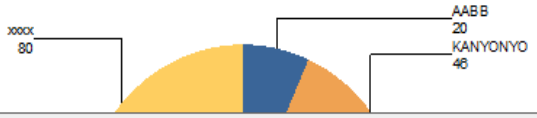
Immeuble Dungu Kinshasa/ Gombe

Direction : Toutes les directions
Grade : Tous les grades

Année : 2 013
Agent : Tous les Agents

STATISTIQUES DES EVALUATIONS DES AGENTS





Nombre total de pages : 3

Facteur de zoom : 100%

CHAP VII. EXERCICES

7.1. LANGAGE C

1. Quels seront les résultats fournis par ce programme ?

```
#include <stdio.h>
main ()
{ int n = 543 ;
int p = 5 ;
float x = 34.5678;
printf ("A : %d %f\n", n, x) ;
printf ("B : %4d %10f\n", n, x) ;
printf ("C : %2d %3f\n", n, x) ;
printf ("D : %10.3f %10.3e\n", x, x) ;
printf ("E : %*d\n", p, n) ;
printf ("F : %*.*f\n", 12, 5, x) ;
}
```

2. Ecrire un programme C servant à calculer la valeur absolue d'un nombre réel x à partir de la définition de la valeur absolue.
3. On souhaite écrire un programme C de résolution dans R de l'équation du second degré : $Ax^2 + Bx + C = 0$
4. On dénomme nombre de Armstrong un entier naturel qui est égal à la somme des cubes des chiffres qui le composent.
5. On souhaite écrire un programme C# de calcul des n premiers nombres parfaits. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs, 1 compris.
6. On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en C# à partir de l'algorithme de la méthode d'Euclide. Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) a et b.
7. On souhaite écrire un programme C# de calcul et d'affichage des n premiers nombres premiers. Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même On opérera une implantation avec des boucles while et do...while.
8. On souhaite écrire un programme C# qui calcule le nombre d'or utilisé par les anciens comme nombre idéal pour la sculpture et l'architecture. Si l'on considère deux suites numériques (U) et (V) telles que pour n strictement supérieur à 2 :

$$U_n = U_{n-1} + U_{n-2} \ et \ V_n = U_n / U_{n-1}$$

On montre que la suite (V) tend vers une limite appelée nombre d'or (nbr d'Or = 1,6180339887498948482045868343656 4).

9. On souhaite écrire un programme C# afin de vérifier sur des exemples, la conjecture de GoldBach (1742), soit : "Tout nombre pair est décomposable en la somme de deux nombres premiers".
10. Ecrire un programme C# implémentant l'algorithme du tri à bulles.
11. Ecrire un programme C# implémentant l'algorithme du tri par insertion.
12. Ecrire un programme C# effectuant une recherche séquentielle dans un tableau linéaire (une dimension) non trié
13. Ecrire un programme C# effectuant une recherche séquentielle dans un tableau linéaire (une dimension) trié avant recherche.
14. effectuer une recherche dichotomique dans un tableau linéaire déjà trié.
15. Quelle est la valeur de i après la suite d'instructions :

```
int i=10;
i = i-(i--);
```

Quelle est la valeur de i après la suite d'instructions :

```
int i=10;
i = i-(--i);
```

16. la priorité des opérateurs

Enlever les parenthèses des expressions suivantes lorsqu'elles peuvent être retirées.

```
a=(25*12)+b;
if ((a>4) &&(b==18)) { }
((a>=6)&&(b<18))||(c!=18)
c=(a=(b+10));
```

Évaluer ces expressions pour a=6, b=18 et c=24. On supposera que les valeurs données le sont pour chacune des lignes : il n'y a pas d'exécution séquentielle comme dans un programme.

17. Écrivez un programme calcul.c qui calcule la distance entre deux points d'un plan :

- Lit les coordonnées de deux points : X_1 (x_1, y_1) et X_2 (x_2, y_2).
- Affiche les données lues
- Calcule la distance d entre les deux points X_1 et X_2, avec la formule :

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

18. Analyse de programme

✓ Soit le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  unsigned char i=7;
  i=i/2; //"/": division entiere...
  switch(i) {
    case 1 : (void)printf("Premier\n");break;
    case 2 : (void)printf("Deuxième\n");break;
    case 3 : (void)printf("Troisième\n");break;
    default : (void)printf("Non classe\n");
  }
  return EXIT_SUCCESS;
}
```

Qu'est ce qui sera affiché à l'écran lors de l'exécution de ce programme ?

✓ Même question pour le programme :

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int i=18;
  i=i-(--i);
  switch(i) {
    case 1 : (void)printf("Premier\n");
    case 2 : (void)printf("Deuxième\n");
    case 3 : (void)printf("Troisième\n");
    default : (void)printf("Non classe\n");
  }
  return EXIT_SUCCESS;
}
```

19. Écrire la partie de programme réalisant la structure de contrôle demandée. (On ne demande pas le programme en entier : pas les "include" pas le main()...)

    Un test sur une valeur dans la variable i : si i vaut 19 on écrit "OK" autrement on écrit "Rejet"

20. Écrire un programme testage.c contenant une fonction main qui :

- lit sur le clavier l'âge de l'utilisateur avec la fonction scanf;
- teste si la réponse est valide par analyse du code retour de scanf et teste si la valeur est comprise entre 0 et 130;
- affiche si l'utilisateur est majeur (>= 18 ans) ou mineur.

21. Soit le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
  int i;
  for (i=0;i<5;i++) {// "/": division entiere
    (void)printf("Module EC%d\n",(i+9)/(i+1));
  }
  return EXIT_SUCCESS;
}
```

Qu'affichera à l'écran l'exécution de ce programme ?

22. Soit le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
  unsigned char i;
  unsigned char tab[5];
//initialisation du tableau
  tab[0]=1;tab[1]=2;tab[2]=4;tab[3]=8;tab[4]=16;
  for (i=0;i<5;i++) {
    (void)printf("Le %d° elt est %d\n",i+1,tab[i]);
  }
  return EXIT_SUCCESS;
}
```

Qu'affichera à l'écran l'exécution de ce programme ?

23. Ecrire un programme *somme* demandant à l'utilisateur de taper 10 entiers et qui affiche leur somme. Le programme ne devra utiliser que 3 variables et ne devra pas utiliser de tableau.

24. Analyse de programme (double boucle)

✓ Soit le programme suivant :

```
#include <stdio.h> // pour printf
#include <stdlib.h> // pour system
main() {
  int i,j;
    system("clear"); //efface ecran sous linux (system("cls"); sous Windows)
    for(i=0;i<5;i++){
      for(j=i;j<5;j++)
        (void)printf("**");
      (void)printf("\n");
    }
    return EXIT_SUCCESS;
}
```

Que sera-t-il affiché à l'écran lors d'une exécution de ce programme ?

✓ Même question pour le programme suivant :

```
#include <stdio.h> // pour printf
#include <stdlib.h> // pour system
main() {
  int i,j;
    system("clear"); //efface ecran sous linux (system("cls"); sous Windows)
    for(i=0;i<5;i++){
      for(j=5-i;j<5;j++)
        (void)printf("++");
      (void)printf("\n");
    }
    return EXIT_SUCCESS;
}
```

25. Triangle d'étoiles (double boucles)

Compléter la fonction afficherTriangle dans le programme ci-dessous : cette fonction devra afficher un triangle rempli d'étoiles (*) sur un nombre de lignes donné passé en paramètre, exemple :

```
*
**
***
```

```
****
*****
******
*******
********
```

- 1ère version : sans utiliser de tableau à l'aide de deux boucles *for* imbriquées.
- 2ème version : avec une seule boucle *for* et un tableau de chaîne de caractère où vous accumulerez des étoiles.

26.

✓ Soit le petit programme suivant :

```
#include <stdio.h>
main()
{
int i, n, som ;
som = 0 ;
for (i=0 ; i<4 ; i++)
{ printf ("donnez un entier ") ;
scanf ("%d", &n) ;
som += n ;
}
printf ("Somme : %d\n", som) ;
}
```

Écrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction for :

- Une instruction while,
- une instruction do... while.

✓ Calculer la moyenne de notes fournies au clavier avec un dialogue de ce type :

```
note 1 : 12
note 2 : 15
```

note 3 : 15
note 4 : 18
note 5 : 20
note 6 : -1
moyenne de ces 5 notes : 16
Le nombre de notes n'est pas connu a priori et l'utilisateur peut en fournir autant qu'il le désire.
Pour signaler qu'il a terminé, on convient qu'il fournira une note fictive négative. Celle-ci ne devra naturellement pas être prise en compte dans le calcul de la moyenne.

27. Écrire un programme nommé argv.c qui affiche :
- ✓ son nom de lancement (argv[0]);
- ✓ le nombre des ces arguments;
- ✓ la valeur de chaque argument reçu.

Rappels : La fonction main d'un programme C reçoit en argument :

- ✓ un entier argc indiquant le nombre d'élément du tableau argv;
- ✓ un tableau de chaines de caractère argv avec :
    - argv[0] : Nom d'appel du programme.
    - argv[i] : Valeur de l'argument de rang i.

28. Écrire un programme *position.c* contenant une fonction principale *main* déterminant si un entier est contenu dans un tableau statique par l'appel à une fonction *position*.

La fonction main :

- ✓ définira et initialisera le tableau d'entier
- ✓ récupèrera dans son tableau d'argument argv le nombre à chercher.
- ✓ appellera la fonction *position*.
- ✓ affichera l'indice de l'élément dans le tableau ou un message indiquant que le nombre n'a pas été trouvé.

La fonction *position* :

- ✓ aura pour prototype : static int position(int t[], int taille, int x).
- ✓ donnera l'indice d'un élément x dans le tableau t, ou -1 si x n'est pas trouvé.

29. Écrire :
    - ✓ une fonction, nommée f1, se contentant d'afficher "bonjour" (elle ne possèdera aucun argument ni valeur de retour),
    - ✓ une fonction, nommée f2, qui affiche "bonjour" un nombre de fois égal à la valeur reçue en argument (int) et qui ne renvoie aucune valeur,
    - ✓ une fonction, nommée f3, qui fait la même chose que f2, mais qui, de plus, renvoie la valeur (int) 0.

30. Écrire un petit programme appelant successivement chacune de ces trois fonctions, après les avoir convenablement déclarées sous forme d'un prototype.

31. Qu'affiche le programme suivant ?

```
int n=5 ;
main()
{
void fct (int p) ;
int n=3 ;
fct(n) ;
}
void fct(int p)
{
printf("%d %d", n, p) ;
}
```

32. Écrire une fonction qui se contente de comptabiliser le nombre de fois où elle a été appelée en affichant seulement un message de temps en temps, à savoir :
    - ✓ au premier appel : *** appel 1 fois ***
    - ✓ au dixième appel : *** appel 10 fois ***
    - ✓ au centième appel : *** appel 100 fois ***
    - ✓ et ainsi de suite pour le millième, le dix millième appel...
    - ✓ On supposera que le nombre maximal d'appels ne peut dépasser la capacité d'un long.

33. Écrire une fonction récursive calculant la valeur de la « fonction d'Ackermann » A définie

pour m>0 et n>0 par :
A(m,n) = A(m-1,A(m,n-1)) pour m>0 et n>0
A(0,n) = n+1 pour n>0

A(m,0) = A(m-1,1) pour m>0

34. Ecrivez un programme majuscule.c qui lit des données sur le flux stdin et écrits sur stdout après avoir transformé les caractères lus en majuscules. Vous utiliserez les fonctions getchar, putchar (stdio.h) et toupper (ctype.h).

Vous testerez votre programme en lui faisant convertir son propre fichier source majuscule.c.

majuscule.exe < majuscule.c

35. Écrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit :
   - ✓ en utilisant uniquement le « formalisme tableau »,
   - ✓ en utilisant le « formalisme pointeur », chaque fois que cela est possible.

36. Écrire une fonction qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers (à un indice) de taille quelconque. Il faudra donc prévoir 4 arguments : le tableau, sa dimension, le maximum et le minimum. Écrire un petit programme d'essai.
37. Écrire une fonction permettant de trier par ordre croissant les valeurs entières d'un tableau de taille quelconque (transmise en argument). Le tri pourra se faire par réarrangement des valeurs au sein du tableau lui-même.
38. Écrire une fonction calculant la somme de deux matrices dont les éléments sont de type double. Les adresses des trois matrices et leurs dimensions (communes) seront transmises en argument.
39. Écrire un programme déterminant le nombre de lettres « e » (minuscules) présentes dans un texte de moins d'une ligne (supposée ne pas dépasser 132 caractères) fourni au clavier.
40. Écrire un programme qui supprime toutes les lettres « e » (minuscules) d'un texte de moins d'une ligne (supposée ne pas dépasser 132 caractères)

fourni au clavier. Le texte ainsi modifié sera créé, en mémoire, à la place de l'ancien.

41. Écrire un programme qui lit au clavier un mot (d'au plus 30 caractères) et qui l'affiche à l'envers.
42. Écrire un programme qui lit un verbe du premier groupe et qui en affiche la conjugaison au présent de l'indicatif, sous la forme :

je chante
tu chantes
il chante
nous chantons
vous chantez
ils chantent

Le programme devra vérifier que le mot fourni se termine bien par « er ». On supposera qu'il ne peut comporter plus de 26 lettres et qu'il s'agit d'un verbe régulier. Autrement dit, on admettra que l'utilisateur ne fournira pas un verbe tel que « manger » (le programme afficherait alors : « nous mangeons »).

43. Écrire un programme qui :

- ✓ lit au clavier des informations dans un tableau de structures du type point défini comme suit :

```
struct point { int num ;
float x ;
float y ;
}
```

Le nombre d'éléments du tableau sera fixé par une instruction #define.

- ✓ affiche à l'écran l'ensemble des informations précédentes.

44. Réaliser la même chose que dans l'exercice précédent, mais en prévoyant, cette fois, une fonction pour la lecture des informations et une fonction pour l'affichage.
45. La fonction fgets de la bibliothèque standard du langage C permet de lire une chaine de caractère de longueur limitée dans un flux.

Vous allez compléter une fonction *lire_ligne* répondant aux spécifications suivantes :

- ✓ Retour d'une ligne lue dans un flux texte passé en paramètre.
- ✓ Vous éliminerez les caractères de saut de ligne lus.
- ✓ La longueur des lignes lues n'est pas limitée.
- ✓ Contrôle des paramètres et retour des codes d'erreurs systèmes, détection de la fin du fichier.
- ✓ Vous utiliserez au maximum les fonctions de la bibliothèque standard du langage C : allocation mémoire, chaines de caractères...
- ✓ Son prototype est donné par *lire_ligne.h*.
- ✓ Vous utiliserez le programme de *main_lire_ligne.c* pour lire_ligne.
- ✓ Vous devrez traiter le fichier *test_lire_ligne.txt* fourni.
- ✓ Les instructions de compilation et d'édition de lien sont dans les commentaires des fichiers fournis.

46. La bibliothèque standard du langage C offre au programmeur plusieurs fonctions pour générer des nombres aléatoires. La plus connue est rand().

    Vous allez écrire un programme *verifrand.c* qui estime la répartition des nombres aléatoires générés : moyenne et dispersion. Nous allons traiter l'ensemble des nombres générés comme une série discrète regroupée.

    a) Générez 1 million (NB_TIRAGE) notes (x_i) entre 0 et 20 (N) comprises (NB_NOTE = 21) à l'aide de la fonction rand().
    b) Répartissez-les dans le tableau effectif (n) où n_i représente l'effectif (nombre d'occurrences cumulées) de la note x_i.
    c) Calculez et affichez la moyenne arithmétique :
    d) calculez et affichez l'écart moyen de la série :

47. Écrire une fonction C calculant la longueur en octets d'une chaîne de caractères, donnée en argument. A titre d'exercice, pas utiliser la fonction strlen() du fichier d'include string.h.

48. Écrire un programme *lgChaine.c* :

- ✓ qui lit des chaines de caractères tapées au clavier (flux stdin);
- ✓ qui calcule la longueur de chaque chaîne entrée et l'affiche ainsi que sa longueur;
- ✓ qui s'arrête si l'utilisateur ne frappe que la touche Entrée ou si le fichier est fini (Ctrl-D tapé par l'utilisateur).

Vous utiliserez :

- ✓ une des fonctions déclarée dans stdio.h.
- ✓ la fonction strlen, ainsi que d'autres si nécessaire, déclarée dans string.h.

49. Soit un texte donné par une chaîne de caractères. Le but est de compter le nombre d'occurrences de chaque lettre sans distinction entre minuscules et majuscules.
50. Déclarer le texte comme un tableau statique initialisé par une chaîne de caractères constante, un tableau d'entiers statique pour compter les occurrences dont la taille est fixée par une constante et un pointeur pour parcourir le texte.
51. Initialiser le vecteur d'entiers avec un parcours par indice.
52. Compter les occurrences en utilisant la conversion entre le type char et le type int (la conversion d'un caractère donne son code dans le standard américain).
53. Donner et expliquer le résultat de l'exécution du programme suivant :

```
#include <stdio.h>
#define taille_max 5

void parcours(int *tab)
{
  int *q=tab;
  do
  {
    printf("%d:%d\n", q-tab, *q-*tab);
  }
  while (++q-tab < taille_max);
}

void bizarre(int **copie, int *source)
{
  *copie=source;
}
int main(void)
{
  int chose[taille_max] = {1,3,2,4,5}, *truc;
  printf("chose : \n");
  parcours(chose);
  bizarre(&truc, chose);
  printf("truc : \n");
```

```
  parcours(truc);

  return 0;
}
```

54. Écrire un fichier source hello.c. Les résultats du programme exécutable seront différents selon les options passées au préprocesseur sur la ligne de commande de compilation. Ce programme affichera :

- "Hello World", si la constante symbolique *WORLD* est définie.
- "Hello Fof", si *FOF* est définie.
- "Hello Nobody", si aucune de ces constantes n'est définie.

Lignes de Compilation avec gcc :

- gcc -D WORLD -o hello.exe hello.c
- gcc -D FOF -o hello.exe hello.c
- gcc -o hello.exe hello.c

Exécution : ./hello.exe

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

int main(void)
{
#if defined(WORLD)
  (void)puts("Hello World");
#elif defined(FOF)
  (void)puts("Hello Fof");
#else
  (void)puts("Hello Nobody\n");
#endif
  return EXIT_SUCCESS;
}
```

55. Écrire un programme permettant d'afficher le contenu d'un fichier texte en numérotant les lignes. Ces lignes ne devront jamais comporter plus de 80 caractères.
56. Écrire un programme permettant de créer séquentiellement un fichier « répertoire » comportant pour chaque personne :

- ✓ nom (20 caractères maximum) ;
- ✓ prénom (15 caractères maximum) ;
- ✓ âge (entier) ;
- ✓ numéro de téléphone (11 caractères maximum).

Les informations relatives aux différentes personnes seront lues au clavier.

57. Écrire un programme permettant, à partir du fichier créé par l'exercice précédent, de retrouver les informations correspondant à une personne de nom donné.
58. Écrire un programme permettant, à partir du fichier créé dans l'exercice 56, de retrouver les informations relatives à une personne de rang donné (par accès direct).

7.2. LANGAGE C#

1. Refaire les exercices de C en C#
2. Un gérant de bibliothèque désire automatiser la gestion des prêts.
   - ✓ Il commande un logiciel permettant aux utilisateurs de connaître les livres présents, d'en réserver jusqu'à 2. L'adhérent peut connaître la liste des livres qu'il a empruntés ou réservés.
   - ✓ L'adhérent possède un mot de passe qui lui est donné à son inscription.
   - ✓ L'emprunt est toujours réalisé par les employés qui travaillent à la bibliothèque. Après avoir identifié l'emprunteur, ils savent si le prêt est possible (nombre max de prêts = 5), et s'il a la priorité (il est celui qui a réservé le livre).

     Ce sont les employés qui mettent en bibliothèque les livres rendus et les nouveaux livres. Il leur est possible de connaître l'ensemble des prêts réalisés dans la bibliothèque.

3. Cette étude de cas concerne un système simplifié de réservation de vols pour une agence de voyages. Les interviews des experts métier auxquelles on a procédé ont permis de résumer leur connaissance du domaine sous la forme des phrases suivantes :

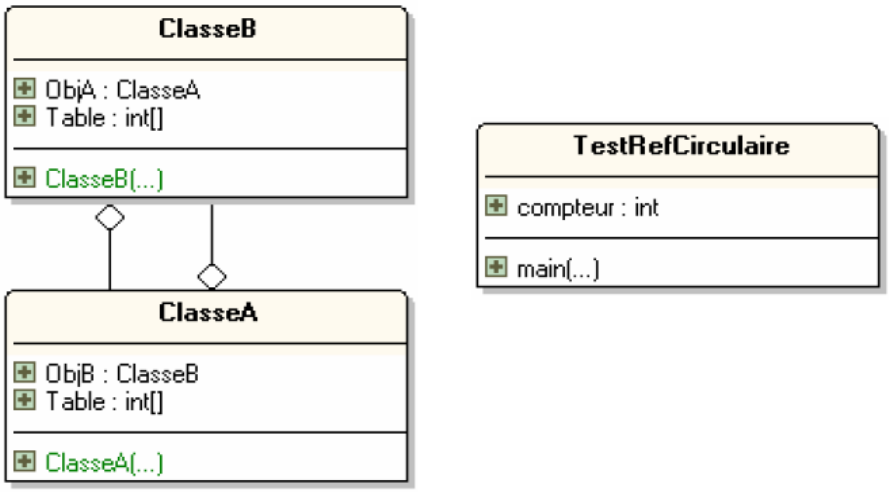
   - ✓ Des compagnies aériennes proposent différents vols.
   - ✓ Un vol est ouvert à la réservation et refermé sur ordre de la compagnie.
   - ✓ Un client peut réserver un ou plusieurs vols, pour des passagers différents.
   - ✓ Une réservation concerne un seul vol et un seul passager.

- ✓ Une réservation peut être annulée ou confirmée.
- ✓ Un vol a un aéroport de départ et un aéroport d'arrivée.
- ✓ Un vol a un jour et une heure de départ, et un jour et une heure d'arrivée.
- ✓ Un vol peut comporter des escales dans des aéroports.
- ✓ Une escale a une heure d'arrivée et une heure de départ.
- ✓ Chaque aéroport dessert une ou plusieurs villes.

Implémentez ces problèmes en C#.

4. Construire un programme permettant de convertir un nombre romain écrit avec les lettres M, D, C, L, X, V, I en un entier décimal.
5. On donne trois classes ClasseA, ClasseB, TestRefCirculaire :
   - ✓ La classe ClasseA possède une référence ObjB à un objet public de classe ClasseB, possède un attribut Table qui est un tableau de 50 000 entiers, lorsqu'un objet de ClasseA est construit il incrémente de un le champ static compteur de la classe TestRefCirculaire et instancie la référence ObjB.
   - ✓ La classe ClasseB possède une référence ObjA à un objet public de classe ClasseA, possède un attribut Table qui est un tableau de 50 000 entiers, lorsqu'un objet de ClasseB est construit il incrémente de un le champ static compteur de la classe TestRefCirculaire et instancie la référence ObjA.
   - ✓ La classe TestRefCirculaire ne possède qu'un attribut de classe : le champ public entier compteur initialisé à zéro au départ, et la méthode principale Main de lancement de l'application.

Implémentez ces trois classes en ne mettant dans le corps de la méthode Main qu'une seule instruction consistant à instancier un objet local de classe ClasseA, puis exécuter le programme et expliquez les résultats obtenus.



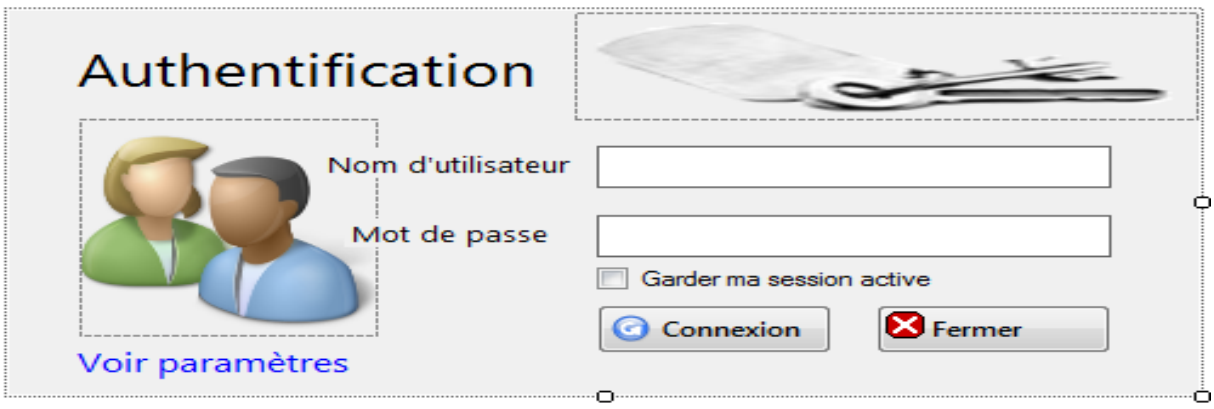
6. Ecrire un programme en C# qui permet de gérer dix premiers clients d'une banque et les sert avec la logique de LIFO.
7. Ecrire un programme en C# qui permet de gérer dix premiers clients d'une banque et les sert avec la logique de FIFO.
8. Ecrire un programme en C# qui permet de stocker les informations estudiantines dans un couplet de matricule et nom.
   A partir d'un numéro matricule, on peut retrouver le nom de l'étudiant.
9. Ecrire un programme C# qui permet de créer et d'enregistrer des informations du type (matricule, nom, post nom, sexe et âge) dans un fichier texte.
   Prévoir des modules permettant de :
   - ✓ Supprimer un étudiant à partir de son numéro matricule
   - ✓ Mettre à jour l'identité d'un étudiant à partir de son numéro matricule
10. Un professeur contient des informations suivantes :
    - ✓ Nom, post nom, prénom, sexe, date de naissance, matricule, département, état civil et spécialité.

    Un étudiant a des informations suivantes :
    - ✓ Nom, post nom, prénom, sexe, date de naissance, matricule, et département.

    Implémentez ces classes sous C# en proposant des méthodes dans chacune de classes et des relations entre classes.
11. Il est question d'implémenter avec C# l'interface ci-dessous :

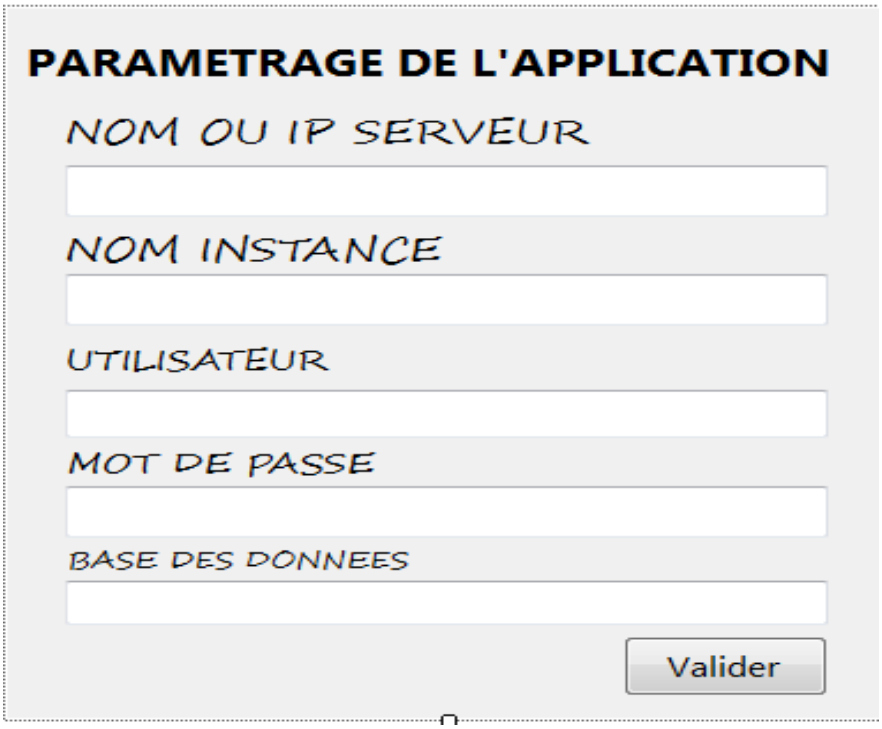
Les informations qui seront saisies dans des rubriques de l'interface seront stockées dans un fichier texte qui portera le nom de « etudiant.txt ».
En cliquant sur le bouton Rechercher, une boite de dialogue s'affiche et demandant une des informations de l'étudiant parmi (son nom, son post nom et son prénom) et le résultat sera afficher dans la liste view.
Par défaut, la liste view contient les informations du fichier.

12. Implémentez cette interface en C# permettant à un utilisateur de s'authentifier en fournissant son login et son mot de passe. Les informations d'authentification sont gérées dans un fichier texte sous le formalisme : login valeur ; password valeur.



13. Proposez des codes C# qui permettent de prendre un flux vidéo à partir d'une webcam d'un ordinateur.
    Voici l'interface proposée :

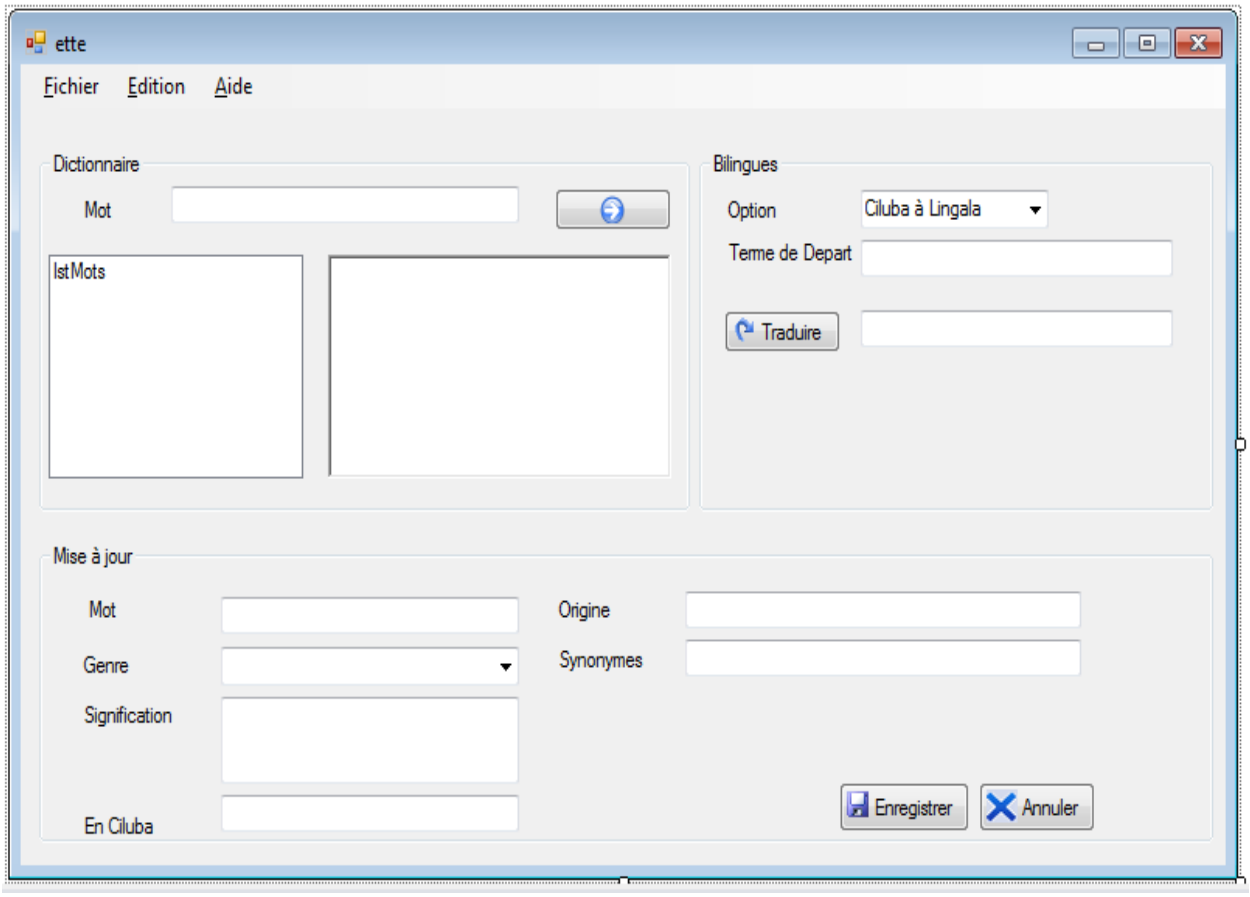
14. Proposez les codes C# permettant de conserver dans des sessions les informations qui seront saisies dans l'interface ci-dessous de sorte qu'à chaque fois qu'on lancera l'interface, les informations antérieurement saisies soient vues via ses contrôles.



15. Proposez des codes C# permettant de simuler un dictionnaire comme décrit dans l'interface ci-dessous.

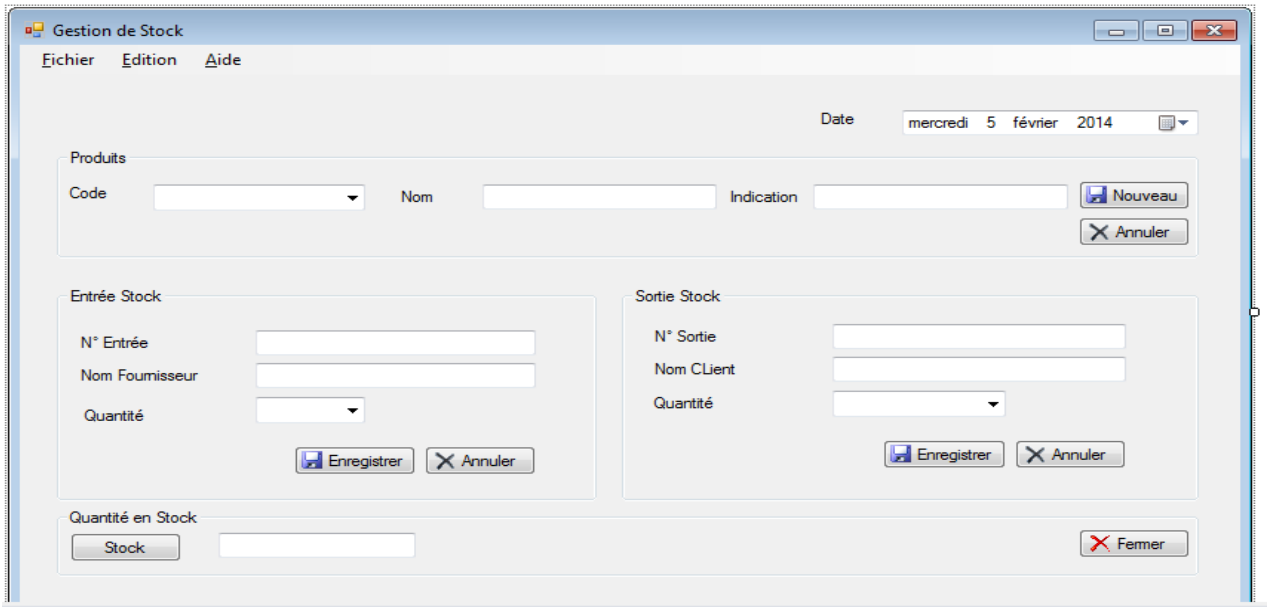
    Le dictionnaire ne contient que deux langues (Swahili et lingala) et à chaque enregistrement, on renseigne le mot, ses synonymes, son genre, sa signification et son origine.

Les informations seront stockées dans un fichier texte.



16. Implémentez l'interface ci-dessous en C#.

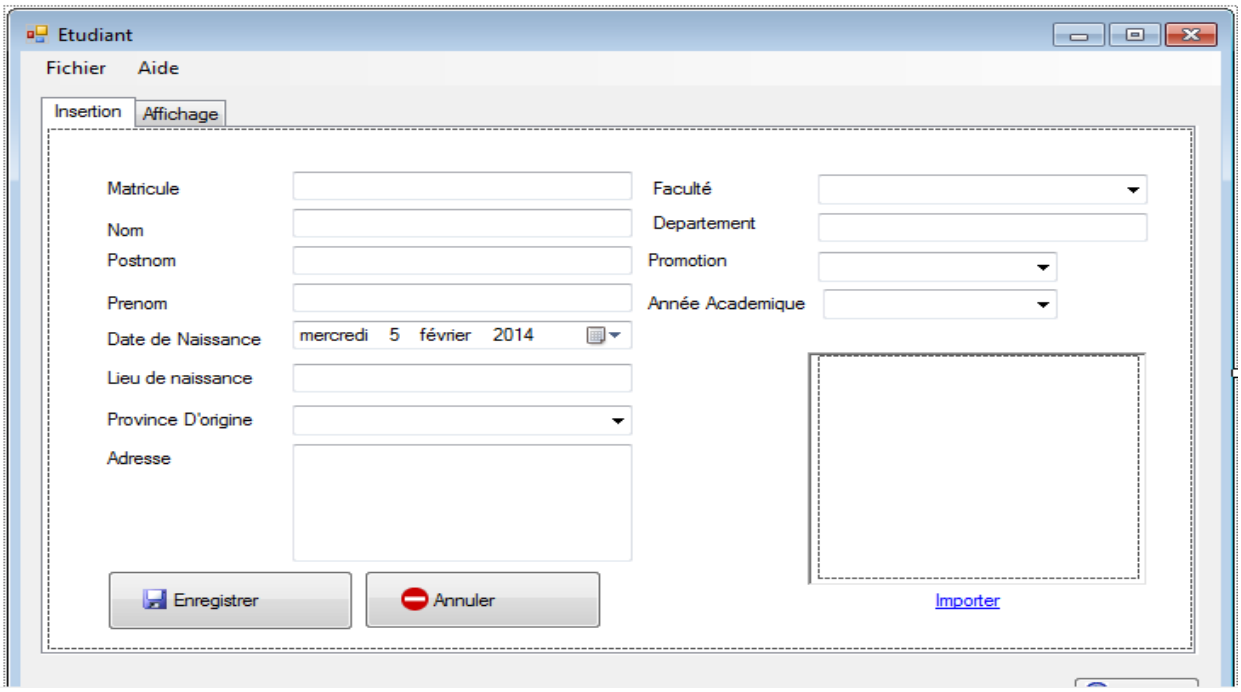
Il s'agit d'une application qui gère le stock d'une entreprise de la place et ses informations seront stockées dans un fichier.

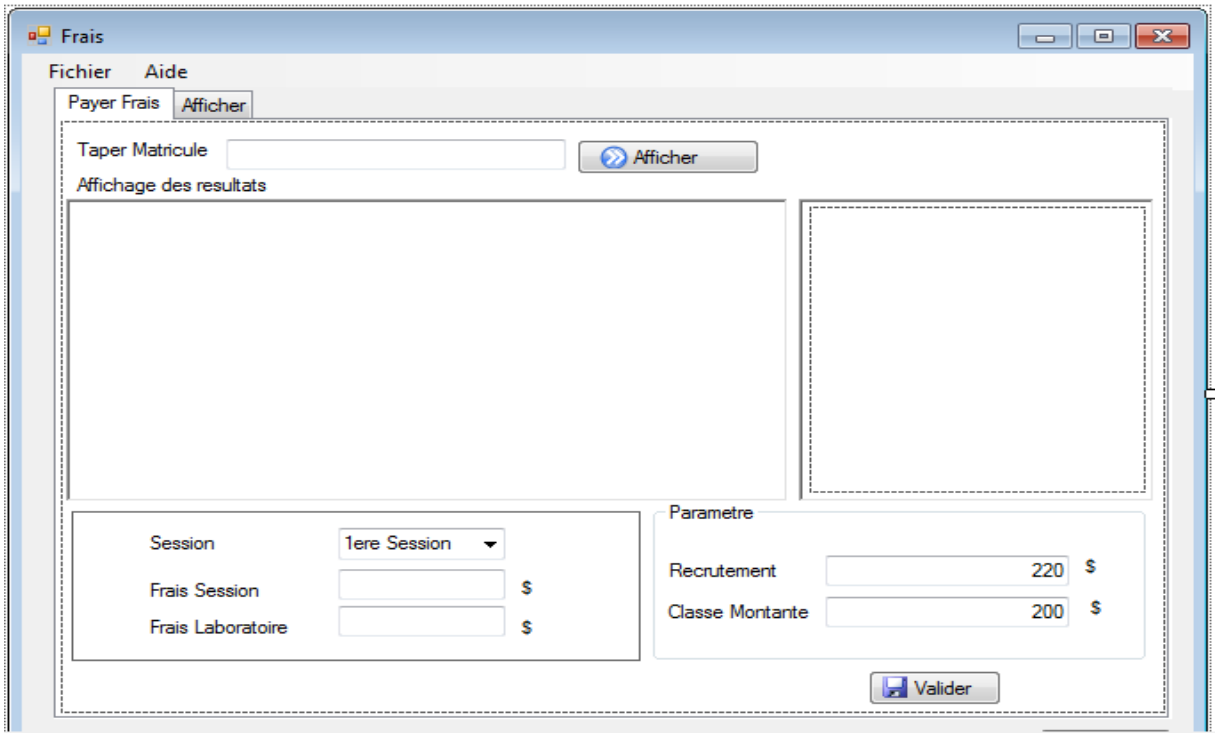


17. Nous comptons via cette nouvelle application de gérer l'inscription et le paiement des frais académiques des étudiants.

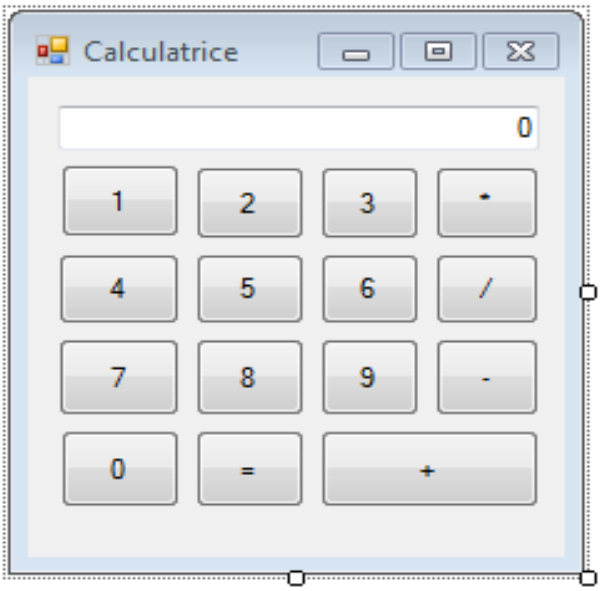
    L'application contient trois interfaces, une pour l'authentification, une deuxième pour l'enregistrement et une troisième pour le paiement de frais.

    Les détails des informations sont repris dans les différentes interfaces ci-dessous :

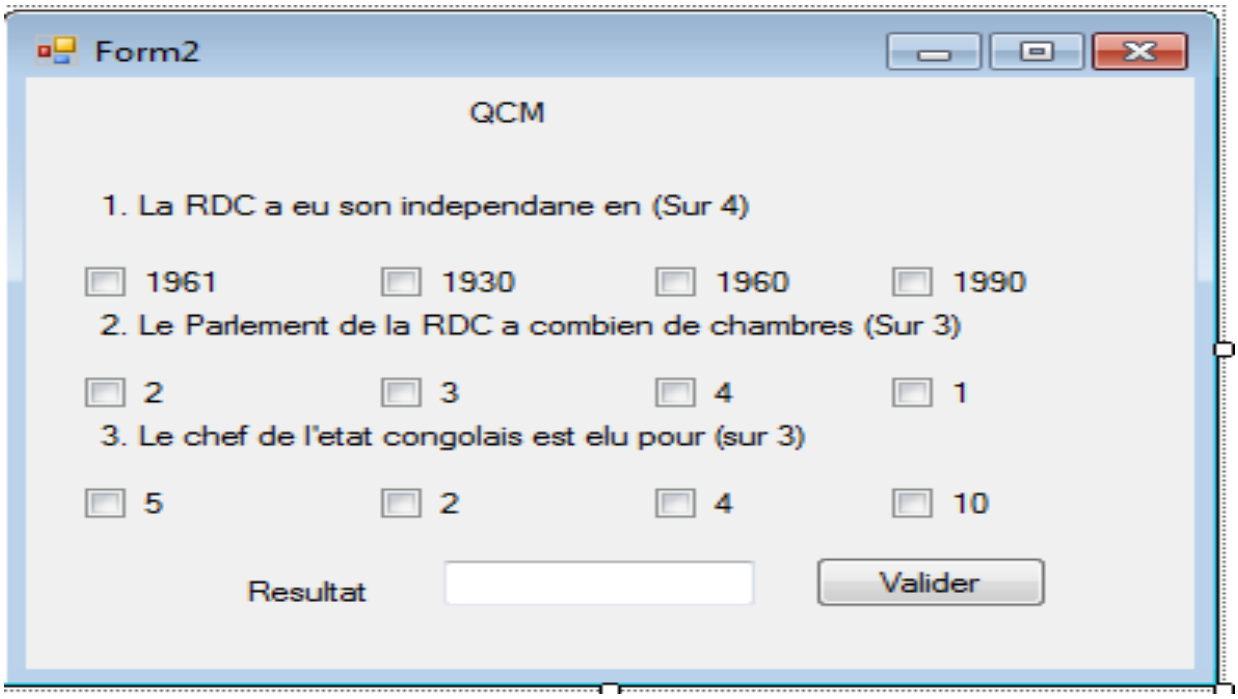




18. Proposez des codes C# pour la manipulation d'une calculatrice.

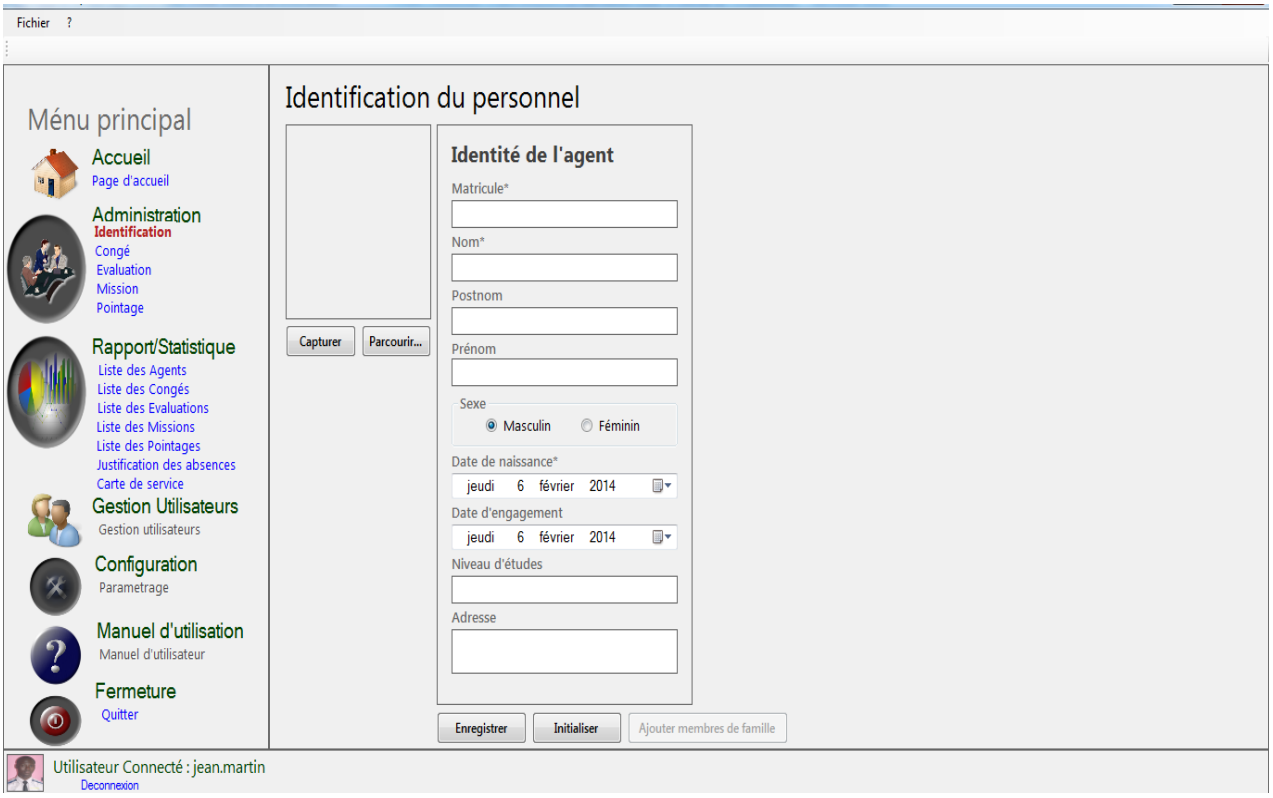


19. Proposez une liste de codes C# permettant à un utilisateur de jouer au jeu illustré à l'interface ci-dessous :


Form2
QCM
1. La RDC a eu son independane en (Sur 4)
1961
1930
1960
1990
2. Le Parlement de la RDC a combien de chambres (Sur 3)
2
3
4
1
3. Le chef de l'etat congolais est elu pour (sur 3)
5
2
4
10
Resultat
Valider

ANNEXES

1. Identification du personnel d'une entreprise



Codes sources

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace ACGT_RH.uc
{
    public partial class Identification : UserControl
    {
        private Entites.Agent _agent = null;

        public Affectation affectation = new Affectation() { Dock = DockStyle.Fill };
        private AffectationView affectationView = new AffectationView() { Dock =
DockStyle.Fill };

        public Identification()
        {
            InitializeComponent();
        }

        public void initialiser()
        {
            matriculeTextBox.Text = string.Empty;
            nomTextBox.Text = string.Empty;
```

```
            postnomTextBox.Text = string.Empty;
            mRadionButton.Checked = true;
            prenomTextBox.Text = string.Empty;
            dateNaissanceDtp.Value = DateTime.Now;
            dateEngagementDtp.Value = DateTime.Now;
            etudesTextBox.Text = string.Empty;
            adressTextBox.Text = string.Empty;
            photoPictureBox.Image = null;

            enregistrerButton.Text = "Enregistrer";

            affectation.initCombobox();
            affectationPanel2.Controls.Clear();
            affectationPanel2.Controls.Add(affectation);

            affectationPanel1.Visible = false;
            famillePanel.Visible = false;
            matriculeTextBox.ReadOnly = false;
            familleButton.Enabled = false;

            infoLibelle.Text = string.Empty;

            Context.Agent = _agent = null;
        }

        public void afficher(Entites.Agent agent)
        {
            if (agent != null)
            {
                matriculeTextBox.Text = agent.CodeAgent;
                nomTextBox.Text = agent.Nom;
                postnomTextBox.Text = agent.Postnom;
                prenomTextBox.Text = agent.Prenom;

                mRadionButton.Checked = agent.Sexe.Equals("M");
                fRadionButton.Checked = agent.Sexe.Equals("F");

                dateNaissanceDtp.Value = agent.DateNaissance;
                dateEngagementDtp.Value =agent.DateEngagement;
                etudesTextBox.Text = agent.NiveauEtude;
                adressTextBox.Text = agent.Adresse;
                photoPictureBox.Image = Utils.GetImageFromBuffer(agent.Photo);

                Entites.Affectation affectation =
DAO.administration.SqlAgent.getAffectationActuelle(agent);
                if (affectation != null)
                {
                    affectationView.afficher(
                        affectation.Direction.Libelle,
                        affectation.Grade.Libelle,
                        affectation.Fonction,
                        affectation.DateAffectation.ToString("dd/MM/yyyy"));

                    affectationPanel2.Controls.Clear();
                    affectationPanel2.Controls.Add(affectationView);

                    affectationButton.Text = "Mise à jour";
                }
                else
                {
                    infoLibelle.Text = "Cet agent n'est affecté à aucune direction
\net ne possède aucun grade ni fonction.";
```

```
                    affectationButton.Text = "Affecter";
                }

                enregistrerButton.Text = "Modifier";
                matriculeTextBox.ReadOnly = true;

                Context.Agent = agent;

                affectationPanel1.Visible = true;
                familleButton.Enabled = true;

                afficherFamille();
            }
        }

        public void afficherFamille()
        {
            if (Context.Agent != null)
            {
                DataTable dt =
DAO.administration.SqlAgent.getMembresfamille(Context.Agent);
                if (dt != null && dt.Rows.Count > 0)
                {
                    familleListBox.DataSource = dt;
                    familleListBox.DisplayMember = "NOM";
                    familleListBox.ValueMember = "CODE_MEMBRE_FAMILLE";

                    famillePanel.Visible = true;
                }
            }
        }

        private void enregistrerButton_Click(object sender, EventArgs e)
        {
            if (matriculeTextBox.Text.Trim().Equals(""))
            {
                MessageBox.Show("Veuillez saisir le matricule de l'agent...");
                return;
            }

            if (nomTextBox.Text.Trim().Equals(""))
            {
                MessageBox.Show("Veuillez saisir le nom de l'agent...");
                return;
            }

            if (_agent == null)
            {
                _agent = new Entites.Agent();
            }

            _agent.CodeAgent = matriculeTextBox.Text.Trim();
            _agent.Nom = nomTextBox.Text.Trim();
            _agent.Postnom = postnomTextBox.Text.Trim();
            _agent.Prenom = prenomTextBox.Text.Trim();
            _agent.Sexe = (mRadionButton.Checked ? "M" : "F");
            _agent.DateNaissance = dateNaissanceDtp.Value;
            _agent.DateEngagement = dateEngagementDtp.Value;
            _agent.NiveauEtude = etudesTextBox.Text.Trim();
            _agent.Adresse = adressTextBox.Text.Trim();
            _agent.Photo = Utils.GetBufferFromImage(photoPictureBox.Image);
```

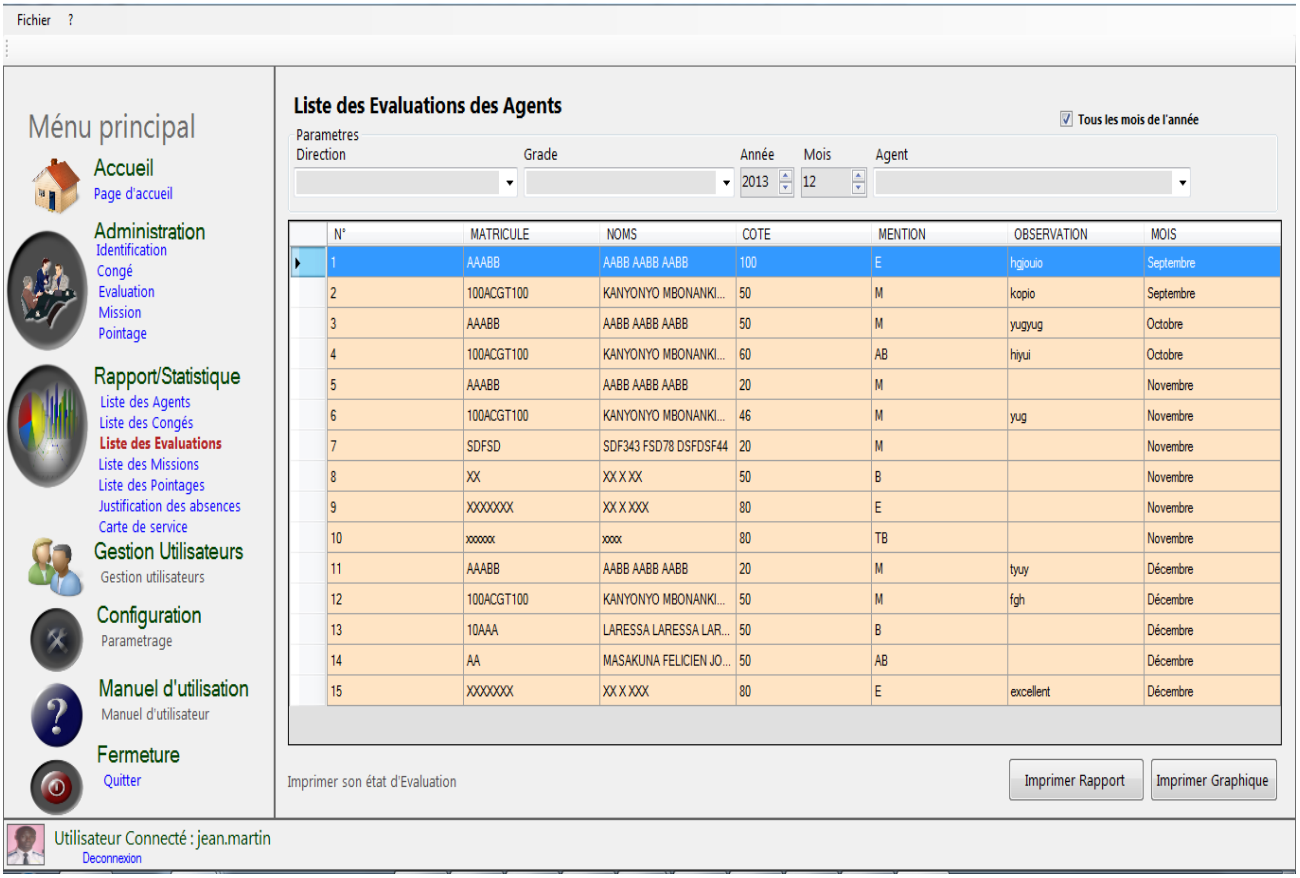
```
            if (enregistrerButton.Text.Equals("Enregistrer"))
            {
                if (MessageBox.Show("Voulez-vous enregistrer ce nouvel agent ?", "",
MessageBoxButtons.YesNo) != DialogResult.Yes)
                {
                    return;
                }

                Entites.Agent agent =
DAO.administration.SqlAgent.getAgent(_agent.CodeAgent);
                if (agent != null)
                {
                    new ui.AgentInfo(agent).ShowDialog();
                    agent = null;
                    return;
                }

                if (DAO.administration.SqlAgent.Enregistrer(_agent))
                {
                    MessageBox.Show("Enregistrement de l'agent avec succès...");

                    Context.Agent = agent;

                    affectation.initCombobox();
                    affectationPanel2.Controls.Clear();
                    affectationPanel2.Controls.Add(affectation);

                    affectationPanel1.Visible = false;

                    infoLibelle.Text = "Cet agent n'est affecté à aucune direction
\net ne possède aucun grade ni fonction.";
                    affectationButton.Text = "Affecter";
                }
                else
                {
                    MessageBox.Show("Echec d'enregistrement de l'agent...");
                }
            }
            else if (enregistrerButton.Text.Equals("Modifier"))
            {
                if (MessageBox.Show("Voulez-vous mettre à jour les informations de cet
agent ?", "", MessageBoxButtons.YesNo) != DialogResult.Yes)
                {
                    return;
                }

                if (DAO.administration.SqlAgent.Modifier(_agent))
                {
                    MessageBox.Show("Mise à jour avec succès...");
                }
                else
                {
                    MessageBox.Show("Echec de mise à jour...");
                }
            }
        }

        private void initialiserButton_Click(object sender, EventArgs e)
        {
            initialiser();
        }
```

```
        private void button1_Click(object sender, EventArgs e)
        {
            ui.CapturePhoto capturePhoto = new ui.CapturePhoto();
            capturePhoto.capturePhotoEvt += new
EventHandler(capturePhoto_capturePhotoEvt);
            capturePhoto.ShowDialog();
        }

        void capturePhoto_capturePhotoEvt(object sender, EventArgs e)
        {
            try
            {
                photoPictureBox.Image = sender as Image;
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        void identification_getAgebtEvent(object sender, EventArgs e)
        {
            _agent = sender as Entites.Agent;
        }

        private void parcourirButton_Click(object sender, EventArgs e)
        {
            try
            {
                if (openFileDialogImage.ShowDialog() == DialogResult.OK)
                {
                    photoPictureBox.Image =
Bitmap.FromFile(openFileDialogImage.FileName);
                }
            }
            catch
            {
                MessageBox.Show("Problème lors du chargement de l'image...");
            }
        }

        private void affectationButton_Click(object sender, EventArgs e)
        {
            if (affectationButton.Text.Equals("Mise à jour"))
            {
                affectation.initCombobox();
                affectationPanel2.Controls.Clear();
                affectationPanel2.Controls.Add(affectation);

                affectationButton.Text = "Affecter";
            }
            else if (affectationButton.Text.Equals("Affecter"))
            {
                if (affectation.directionComboBox.SelectedValue == null |
                    affectation.gradesComboBox.SelectedValue == null)
                {
                    MessageBox.Show("Veuillez sélectionner la direction...");
                    return;
                }

                if (affectation.directionComboBox.SelectedIndex == 0 |
                    affectation.gradesComboBox.SelectedIndex == 0)
```

```
                {
                    MessageBox.Show("Veuillez sélectionner le grade...");
                    return;
                }

                if (affectation.fonctionTextBox.Equals(""))
                {
                    MessageBox.Show("Veuillez saisir la fonction...");
                    return;
                }

                Entites.Affectation affectationAgent = new Entites.Affectation()
                {
                    Agent = Context.Agent,
                    Direction = new Entites.Direction()
                    {
                        CodeDirection =
(int)affectation.directionComboBox.SelectedValue,
                        Libelle = affectation.directionComboBox.Text
                    },
                    Grade = new Entites.Grade()
                    {
                        CodeGrade = (int)affectation.gradesComboBox.SelectedValue,
                        Libelle = affectation.gradesComboBox.Text
                    },
                    Fonction = affectation.fonctionTextBox.Text,
                    DateAffectation = affectation.dateAffectationDtp.Value,
                };

                if (DAO.administration.SqlAgent.affectation(affectationAgent))
                {
                    MessageBox.Show("Affectation de l'agent avec succès...");

                    if (affectationAgent != null)
                    {
                        affectationView.afficher(
                            affectationAgent.Direction.Libelle,
                            affectationAgent.Grade.Libelle,
                            affectationAgent.Fonction,
                            affectationAgent.DateAffectation.ToString("dd/MM/yyyy"));

                        affectationPanel2.Controls.Clear();
                        affectationPanel2.Controls.Add(affectationView);

                        affectationButton.Text = "Mise à jour";
                        infoLibelle.Text = string.Empty;
                    }
                }
                else
                {
                    MessageBox.Show("Echec d'affectation de l'agent...");
                }
            }
        }

        private void familleButton_Click(object sender, EventArgs e)
        {
            new ui.MembreFamille().ShowDialog();

            if (ui.ConstitutionFamille.MembreFamilleAajouter != null)
            {
                new ui.ConstitutionFamille().ShowDialog();
```

```
                if (ui.ConstitutionFamille.MembreFamilleAajouter != null)
                {
                    if (DAO.administration.SqlAgent.AjouterMembreFamille
                        (Context.Agent, ui.ConstitutionFamille.MembreFamilleAajouter,
ui.ConstitutionFamille.TypeMembre))
                    {
                        afficherFamille();
                    }
                }
            }
        }
        private void familleListBox_DoubleClick(object sender, EventArgs e)
        {
            if (familleListBox.SelectedValue != null)
            {
                string codeMembreFamille = (string)familleListBox.SelectedValue;
                Entites.MembreFamille membreFamille =
                    DAO.administration.SqlFamille.getMembreFamille(codeMembreFamille);

                new ui.VisualisationMembreFamille(membreFamille).ShowDialog();
            }
        }

    }
}
```

2. Liste des évaluations des agents d'une entreprise



Codes sources

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using CrystalDecisions.CrystalReports.Engine;

namespace ACGT_RH.uc
{
    public partial class UCListeEvaluation : UserControl
    {
        SortedList<int, string> direction = null;
        SortedList<int, string> listeGrade = null;
        SortedList<string, string> listeAgent = null;
        public UCListeEvaluation()
        {

            InitializeComponent();
            DataGridViewRow row = dgvListeEvaluation.RowTemplate;
            row.DefaultCellStyle.BackColor = Color.Bisque;
            row.Height = 25;
            row.MinimumHeight = 20;

            direction = DAO.administration.SqlEvaluation.GetDirections();
            IEnumerator<string> libelle = direction.Values.GetEnumerator();
            cbDirection.Items.Clear();
            cbDirection.Items.Add("");
            while (libelle.MoveNext())
            {
                cbDirection.Items.Add(libelle.Current);
            }
            DateTime date = DateTime.Now;
            numAnnee.Value = date.Year;
            cbDirection.Text = "";
        }

        private void dgvListeEvaluation_CellContentClick(object sender,
DataGridViewCellEventArgs e)
        {

        }

        private void cbDirection_SelectedIndexChanged(object sender, EventArgs e)
        {

            try
            {
                listeGrade = DAO.administration.SqlEvaluation.GetListeGrade();
                IEnumerator<string> libelle = listeGrade.Values.GetEnumerator();
                cbGrade.Items.Clear();
                cbGrade.Items.Add("");
                while (libelle.MoveNext())
                {
                    cbGrade.Items.Add(libelle.Current);
                }
                cbGrade.SelectedIndex = 0;
                remplirAgents(cbDirection.Text, cbGrade.Text);
                remplirTableau();
            }
```

```
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }

        }
        private void remplirTableau()
        {
            try
            {
                DataTable table = null;
                if (checkBoxEtatMois.Checked == true)
                {
                    if (cbAgent.Text.Trim().Equals(""))
                    {
                        table =
DAO.administration.SqlEvaluation.GetAgents(cbDirection.Text, cbGrade.Text, "",
(int)numAnnee.Value);
                    }
                    else
                    {
                        table =
DAO.administration.SqlEvaluation.GetAgents(cbDirection.Text, cbGrade.Text,
listeAgent.Keys.ElementAt(cbAgent.SelectedIndex - 1), (int)numAnnee.Value);
                    }
                }
                else
                {
                    if (cbAgent.Text.Trim().Equals(""))
                    {
                        table =
DAO.administration.SqlEvaluation.GetAgents(cbDirection.Text, cbGrade.Text, "",
(int)numAnnee.Value, (int)numMois.Value);
                    }
                    else
                    {
                        table =
DAO.administration.SqlEvaluation.GetAgents(cbDirection.Text, cbGrade.Text,
listeAgent.Keys.ElementAt(cbAgent.SelectedIndex - 1), (int)numAnnee.Value,
(int)numMois.Value);
                    }

                }

                dgvListeEvaluation.Rows.Clear();
                if (table != null)
                {
                    for (int i = 0; i < table.Rows.Count; i++)
                    {
                        dgvListeEvaluation.Rows.Add(1);
                        dgvListeEvaluation.Rows[i].Cells["N"].Value = ""+(i+1);
                        dgvListeEvaluation.Rows[i].Cells["MATRICULE"].Value =
table.Rows[i].Field<string>("CODE_AGENT");
                        dgvListeEvaluation.Rows[i].Cells["NOMS"].Value =
table.Rows[i].Field<string>("NOM") + " " + table.Rows[i].Field<string>("POSTNOM") + "
" + table.Rows[i].Field<string>("PRENOM");
                        dgvListeEvaluation.Rows[i].Cells["COTE"].Value =
table.Rows[i].Field<int>("COTE");
                        dgvListeEvaluation.Rows[i].Cells["MENTION"].Value =
table.Rows[i].Field<string>("MENTION");
                        dgvListeEvaluation.Rows[i].Cells["OBSERVATION"].Value =
table.Rows[i].Field<string>("OBSERVATION");
```

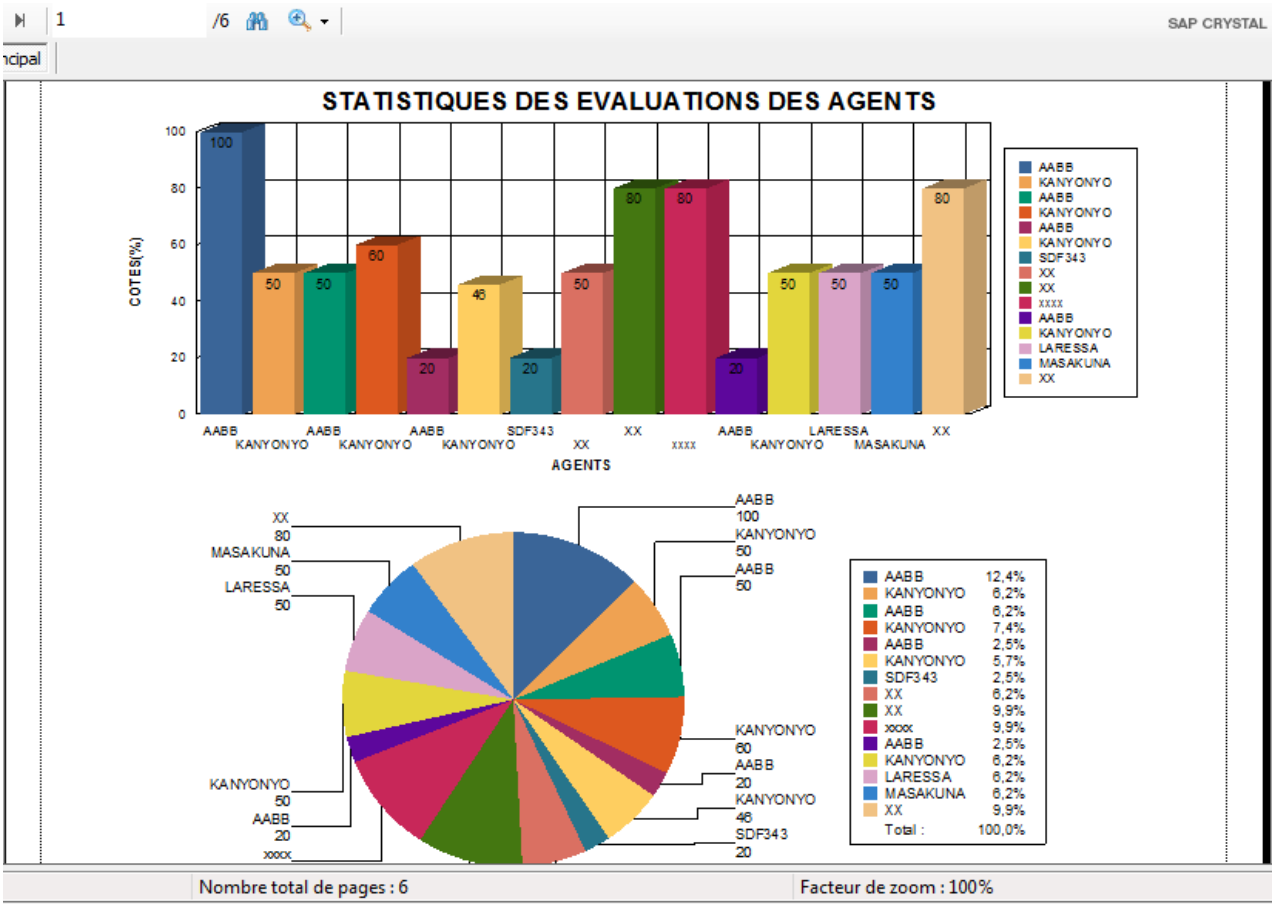
```
                        dgvListeEvaluation.Rows[i].Cells["MOIS"].Value =
NumericToStringMois(table.Rows[i].Field<int>("MOIS"));
                    }
                }
            }
            catch
            {
            }


        }

        private void numAnnee_ValueChanged(object sender, EventArgs e)
        {
            remplirTableau();
        }

        private void cbAgent_SelectedIndexChanged(object sender, EventArgs e)
        {
            remplirAgents(cbDirection.Text, cbGrade.Text);
            remplirTableau();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                string direction = cbDirection.Text;
                string grade = cbGrade.Text;
                string codeAgent = null;
                 if(cbAgent.Text.Trim().Equals("")){
                     codeAgent = "";
                }
                else{
                    codeAgent=listeAgent.Keys.ElementAt(cbAgent.SelectedIndex-1);
                }

                int Annee = (int)numAnnee.Value;
                int Mois = (int)numMois.Value;
                if (checkBoxEtatMois.Checked == true)
                {
                    Mois = 0;
                }
                ui.UIRapportViewer visionneur = new ui.UIRapportViewer(direction,
grade, codeAgent, Annee,Mois,"liste");
                visionneur.ShowDialog();
            }
            catch(Exception exception)
            {
                MessageBox.Show(exception.Message);
            }


        }

        private void UCListeEvaluation_Load(object sender, EventArgs e)
        {

        }
        private void remplirAgents(string direction, string grade)
        {
            try
```

```
            {
                cbAgent.Items.Clear();
                cbAgent.Items.Add("");
                cbAgent.Text = "";
                if (direction.Trim().Equals("") && grade.Trim().Equals(""))
                {
                    listeAgent = DAO.administration.SqlEvaluation.GetListeAgents();
                }
                else
                {
                    if (direction.Trim().Equals("") && (!grade.Trim().Equals("")))
                    {
                        listeAgent =
DAO.administration.SqlEvaluation.GetListeAgents(grade);
                    }
                    else
                    {
                        if (!(direction.Trim().Equals("")) && grade.Trim().Equals(""))
                        {
                            listeAgent =
DAO.administration.SqlEvaluation.GetListeAgentparDirection(direction);
                        }
                        else
                        {
                            listeAgent =
DAO.administration.SqlEvaluation.GetListeAgents(direction, grade);
                        }
                    }
                }


                IEnumerator<string> noms = listeAgent.Values.GetEnumerator();

                while (noms.MoveNext())
                {
                    cbAgent.Items.Add(noms.Current);
                }
                cbAgent.SelectedIndex = 0;
            }
            catch
            {

            }

        }

        private void cbAgent_SelectedIndexChanged_1(object sender, EventArgs e)
        {
            remplirTableau();
            if (cbAgent.Text.Trim().Equals(""))
            {
                linkLabelEvaluation.Enabled = false;
            }
            else
            {
                linkLabelEvaluation.Enabled = true;
            }
        }

        private void btSup_Click(object sender, EventArgs e)
        {
            try
```

```
            {
                string direction = cbDirection.Text;
                string grade = cbGrade.Text;
                string codeAgent = null;
                if (cbAgent.Text.Trim().Equals(""))
                {
                    codeAgent = "";
                }
                else
                {
                    codeAgent = listeAgent.Keys.ElementAt(cbAgent.SelectedIndex - 1);
                }

                int Annee = (int)numAnnee.Value;
                int Mois = (int)numMois.Value;
                if (checkBoxEtatMois.Checked == true)
                {
                    Mois = 0;
                }
                ui.UIRapportViewer visionneur = new ui.UIRapportViewer(direction,
grade, codeAgent, Annee, Mois, "graphique");
                visionneur.ShowDialog();
            }
            catch (Exception exception)
            {
                MessageBox.Show(exception.Message);
            }
        }

        private void checkBoxEtatMois_CheckedChanged(object sender, EventArgs e)
        {
            remplirTableau();
        }

        private void numMois_ValueChanged(object sender, EventArgs e)
        {
            remplirTableau();
        }
        private string NumericToStringMois(int numeroMois)
        {
            string stringMois = "";
            if (numeroMois == 1)
            {
                stringMois = "janvier";
            }
            else if (numeroMois == 2)
            {
                stringMois = "Février";
            }
            else if (numeroMois == 3)
            {
                stringMois = "Mars";
            }
            else if (numeroMois == 4)
            {
                stringMois = "Avril";
            }
            else if (numeroMois == 5)
            {
                stringMois = "Mai";
            }
            else if (numeroMois == 6)
```

```
            {
                stringMois = "Juin";
            }
            else if (numeroMois == 7)
            {
                stringMois = "Juillet";
            }
            else if (numeroMois == 8)
            {
                stringMois = "Août";
            }
            else if (numeroMois == 9)
            {
                stringMois = "Septembre";
            }
            else if (numeroMois == 10)
            {
                stringMois = "Octobre";
            }
            else if (numeroMois == 11)
            {
                stringMois = "Novembre";
            }
            else
            {
                stringMois = "Décembre";
            }
            return stringMois;
        }

        private void linkLabelEvaluation_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
        {
            ui.UIRapportViewer visionneur = new
ui.UIRapportViewer(listeAgent.Keys.ElementAt(cbAgent.SelectedIndex - 1),
(int)numAnnee.Value);
            visionneur.ShowDialog();
        }
    }
}
```

3. Etat de sortie des évaluations


SAP CRYSTAL
STATISTIQUES DES EVALUATIONS DES AGENTS
COTES(%)
AGENTS
AABB
KANYONYO
SDF343
XX
xxxx
LARESSA
MASAKUNA
Nombre total de pages : 6
Facteur de zoom : 100%

BIBLIOGRAPHIE

1. **Claude Delanoy**, *Programmer en langage C*, 5ème édition 2009 Eyrolles
2. **Gérard Leblanc**, *C# et .NET versions 1 à 4*, éditions Eyrolles
3. **Jessee Michaël C et Edouard**, *Initiation au langage C*, 2008
4. **Mbuyi Mukendi E.**, *Langage scientifique C*, 2008, Cours inédit, UNIKIN

TABLE DES MATIERES