Licence SHS
Université Bordeaux 2
Support du cours ASD

Algorithmique et Structures de Données avec Delphi

Jean Marc Salotti

Résumé

Intitulé	**ASD : Algorithmes et structures de données, plateforme Delphi**		***Langue : Français***
Crédits : 3	Code :	Responsable : Jean Marc Salotti	
Cours : 12 h	Heures/semaine : 1	Total semaines : 12	Semestre : 5
T D : 18 h	Heures/semaine : 1,5	Total semaines : 12	Semestre : 5
Pré requis	Introduction à la programmation, algorithmique de base		
Contenu	Révisions sur Pascal et Delphi. Introduction au langage objet Delphi, à la programmation événementielle, aux composants graphiques de Delphi, utilisation des structures de données listes, piles, manipulation de fichiers textes .ini		
Objectifs	Conception structurée de programmes avec Delphi		
Méthode	Cours ; exercices sur machine en TD		
Évaluation	examen écrit		

Sommaire

INTRODUCTION

Pourquoi Delphi :

- parce que le temps est révolu de "programmer dans un langage", on développe des applications informatiques avec une "plateforme de développement".
- parce que Delphi permet la programmation événementielle
- parce que le langage Pascal Objet est simple et pédagogiquement acceptable
- parce que la programmation sous Delphi est très rapide
- parce que Delphi permet la conception d'interfaces professionnelles
- parce que Delphi est gratuit
- parce qu'il existe un grand nombre de programmeurs Delphi dans le monde et que de nombreux composants sont proposés sur le web pour faire par exemple de la reconnaissance vocale, pour récupérer l'image d'un scanner ou d'une caméra, etc.

Exemples d'applications présentées en cours et nécessitant très peu de code :
- reconnaissance vocale
- synthèse vocale, agents animés (Microsoft Agent)
- live vidéo et capture d'image en utilisant une webcam

Autres applications présentées en cours :
- traitement d'images
- synthèse d'images
- mouvements elliptiques de planètes et astéroïdes autour du soleil
- gestion d'arbre généalogique
- jeu de simulation de type Age of Empires : Marsbase

Connaissances requises pour suivre le cours ASD :
Bases du langage Pascal (un polycopié en ligne est disponible pour ceux qui ont besoin d'une mise à niveau, avec cours et exercices corrigés).

Justification de la maîtrise d'une plateforme de développement :
Les logiciels du commerce ne font pas tout ce qu'on voudrait faire. Il est donc nécessaire d'apprendre à créer soi-même les applications informatiques qui relèvent du domaine des mathématiques appliquées aux sciences humaines et sociales (SHS).
Dans le domaine de la recherche, on peut également noter l'application des principaux concepts et méthodes utilisés dans les modèles de ***systèmes complexes adaptatifs*** proposés en sciences humaines et sociales (travaux des sociologues, politologues, historiens, ou des économistes Axelrod, Bowles et Gintis, Epstein et Axtell, Schelling). Ils portent sur l'émergence de la communication et du langage, de la coopération, la sélection des normes et l'émergence des institutions, l'étude de systèmes économiques, en particulier des marchés et réseaux de production et d'échange, ou encore sur l'organisation sociale de l'espace.
Parmi les concepts et méthodes développés dans ce domaine de recherche figurent les notions d'émergence, d'attracteurs, de lois d'échelle, de régime dynamique, et de transition, de percolation, d'apprentissage, de jeux évolutionnaires etc., dont la modélisation nécessite souvent le développement d'applications informatiques dédiées.

1. Rappels sur le langage Pascal

1.1 Syntaxe générale

Pas de différenciation minuscule / majuscule,
Les commentaires commencent par /* et finissent par */ ou commencent par (* et par *)
Sur une ligne, ce qui est après // est considéré comme commentaires.
L'espace et le retour à la ligne sont des séparateurs. Il faut au moins 1 séparateur entre chaque mot-clé ou variable.

1.2 Types prédéfinis

Il existe 5 grands types de base :
integer, real, boolean = {true, false}, char, string
En cas de nécessité, les entiers et les réels peuvent toutefois se décliner en :
byte, longint, currency et double
La déclaration d'un tableau statique se fait de la façon suivante :
array[indice1..indice2] of nom_de_type

1.3 Création de type

La création d'un nouveau type est précédée du mot-clé "type".
Exemples :

```
type        ttab = array[1..100] of real;
            entier = integer;
```

La déclaration d'une structure composée nécessite le mot clé record et se termine par end;
Exemple : rpoint = record x,y : real; end;

1.4 Instructions de base

Remarques préliminaires :
Dans les descriptions ci-dessous, une expression est une formule mathématique quelconque (par exemple 3*x+1 ou sin(4)) qui aura une valeur à l'exécution du programme, alors qu'une instruction est une opération qui exploite des expressions et éventuellement d'autres instructions mais qui n'a aucune valeur à l'exécution.

- **Instruction d'affectation :**

nom_de_variable := expression // La variable prend la valeur de l'expression

- **Instruction conditionnelle :**

if expression booléenne **then** instruction1 **else** instruction2 // Jamais de ; avant else
Le **else** est optionnel. Si plusieurs **if then else** sont imbriqués et qu'il y a ambiguïté, le **else** se rapporte toujours au **then** le plus proche qui le précède.

- **Instruction "tant que" :**

while expression booléenne **do** instruction
Tant que l'expression booléenne est évaluée à *true* l'instruction après **do** est répétée.

- **Instruction composée :**

begin instruction1; instruction2; instruction3; … **end**;

- **Instruction "répéter jusqu'à" :**

repeat instruction **until** expression booléenne

- **Instruction d'itération "pour" :**

for nom_variable := expression1 **to** expression2 **do** instruction // variable non réelle

est équivalent à :

```
nom_de_variable := expression1;
while nom_de_variable <= expression2 do
begin
        instruction;
        inc(nom_de_variable);          // Augmentation de 1 si c'est un entier
end;
```

Variante descendante : **for** nom_variable := expression1 **downto** expression2 **do** instruction

- **Instruction du traitement cas par cas :**

```
case expression of :
        valeur1 : instruction1;
        valeur2 : instruction2;
        …
        else instructionN;
end;
```

Notez que l'affectation, l'instruction conditionnelle, le while et l'instruction composée suffisent à écrire n'importe quel programme. D'autres instructions moins fondamentales existent, se rapporter au manuel utilisateur ou à l'aide en ligne.

1.5 Exercices de révision sur les boucles

Dans les exercices suivants, on utilisera les fonctions StrToInt pour convertir une chaine de caractères en entier, IntToStr pour la conversion inverse, FloatToStr pour convertir un réel en string et StrToFloat pour la conversion inverse. On utilisera également la méthode TMemo.lines.add(texte) pour afficher une chaîne de caractères dans un TMemo.

1. Définir les variables et écrire la suite d'instructions permettant de calculer la somme de 10 nombres pairs consécutifs, en commençant à partir de 4. Le résultat est placé dans une variable appelée "res".

Solution 1 :

```
var i, pair, res : integer;
begin
        res := 0;
        pair := 4;
        for i:=1 to 10 do       // 10 passages dans la boucle, car il y a 10 calculs à faire
        begin
                res := res + pair;
                pair := pair + 2;
        end;
end;
```

Solution 2 :

```
var i, res : integer;
begin
        res := 0;
```

```
        for i:=1 to 10 do       // toujours 10 passages dans la boucle !!!
                res := res + i*2 + 2;  // Formule en fonction de i
end;
```

2. Ecrire un programme qui demande une valeur de n à l'utilisateur dans un TEdit et qui affiche les n premiers nombres impairs par ordre décroissant dans un TMemo.

3. Soit la suite définie par $\{U_1 = 10; U_{n+1}= 2.U_n - 3\}$. Ecrire un programme qui demande une valeur de n à l'utilisateur dans un TEdit et qui affiche les n premiers membres de cette suite dans un TMemo.

4. Ecrire un programme qui demande à l'utilisateur une valeur de n dans un TEdit et qui affiche le résultat de n! dans un TLabel.

5. Ecrire un programme qui demande à l'utilisateur une valeur pour un réel x et un entier y positif dans 2 TEdit et qui affiche le résultat de x^y dans un TLabel.

6. Ecrire un programme qui demande à l'utilisateur une valeur de n dans un TEdit puis calcule la somme des n premiers carrés ($1 + 4 + 9 + 16 + 25\ldots + n^2$) en affichant tous les résultats intermédiaires dans un TMemo.

7. Montrez l'évolution du contenu des variables x et y, puis déduire le résultat à l'écran des instructions suivantes :

```
var     x,y : integer;
begin
        x := 0; y :=1;
        while (x <=2) do
        begin
                x := x + 1;     y := y + x;
        end;
        form1.memo1.lines.add ('La valeur finale de y est : ' + y);
end;
```

8. Recherche dichotomique. Ecrire un programme qui calcule l'abscisse du point d'intersection entre les courbes y= ln(x) et y=1/x avec une précision de 0,001. La méthode consiste à choisir 2 points d'abscisses x1, x2 autour de l'intersection et à situer le point milieu. Selon sa position, on l'échange avec un des 2 points ce qui permet de diviser l'intervalle par 2. Ensuite on recommence. On pourra exploiter le fait qu'à gauche de l'intersection $\ln(x)<1/x$ et qu'à droite, au contraire, $\ln(x)>1/x$.

9. Jeu du devine le nombre : écrire un programme qui crée un nombre entier aléatoire x entre 0 et 9999 et qui demande à un joueur humain d'essayer de le trouver. A chaque essai du joueur, l'ordinateur doit répondre si celui-ci est "trop grand", "trop petit" ou s'il a "gagné". Le jeu se poursuit tant que l'être humain n'a pas trouvé le nombre x. Lorsque le joueur a gagné, l'ordinateur annonce le nombre d'essais effectués par le joueur. L'ordinateur demande ensuite au joueur s'il veut rejouer et agit en conséquence.
Il existe en Pascal un générateur de nombres aléatoires. Pour l'utiliser, il faut écrire l'instruction randomize au début du programme (formcreate par exemple). Ensuite, la fonction random(expression) a pour valeur un nombre réel compris entre 0 et la valeur de l'expression.
Exemple : x := random (100); { Donne à x une valeur dans l'intervalle [0,100[}

2. Révision sur les booléens et les tableaux

2.1 Les booléens

Les booléens sont utilisés pour exprimer qu'une condition est vraie ou fausse. Le type booléen est défini par un ensemble constitué de 2 éléments { true; false } et obéit à « l'algèbre de Boole ».
De même que pour les integer ou les real, il existe des opérateurs faisant intervenir les booléens :

• Le and
Le **and** est un opérateur binaire qui nécessite donc deux opérandes. Un **and** entre 2 expressions booléennes a et b permet d'exprimer qu'une condition est respectée (vraie) si et seulement si les conditions définie par a **et** b sont respectées (vraies). De même qu'il existe une table d'addition ou de multiplication pour les entiers, il existe une table du **and** pour les booléens :

AND	**false**	**true**
false	false	false
true	false	true

Exemple :
Hypothèses :
- A est un booléen qui vaut true si Pierre est plus grand que Paul et false sinon
- B est un booléen qui vaut true si Pierre est plus grand que François et false sinon
(A and B) vaut true si Pierre est plus grand que Paul **et** Pierre est plus grand que François, et vaut false dans tous les autres cas, ce qui correspond à notre logique intuitive.

• Le or
Le **or** est également un opérateur binaire. Un **or** entre 2 expressions booléennes a et b permet d'exprimer qu'une condition est respectée (vraie) si et seulement si l'une **ou** l'autre des deux conditions définies par a et b est respectée (vraie). La table du **or** est présentée ci-dessous.

OR	**false**	**true**
false	false	true
true	true	true

Reprenons notre exemple :
(A or B) vaut true si Pierre est plus grand que Paul **ou** Pierre est plus grand que François **ou** les deux, et vaut false sinon, ce qui correspond là encore à notre logique intuitive.

• Le not
Le not est le troisième opérateur booléen. Il exprime tout simplement la valeur contraire d'un booléen. Ainsi, x étant un booléen, si x vaut true, not x vaut false et inversement, si x vaut false, not x vaut true.

Enfin, il ne faut pas oublier les opérateurs de comparaison entre entier ou réels dont le résultat est un booléen. Par exemple (10 > 5) vaut true et (5 = 6) vaut false. On peut bien entendu

combiner les opérateurs : si x, y et z sont des réels, l'expression (x > y) and (x > z) est un booléen qui vaut true si et seulement si x est supérieur à y et z.

2.2 Exercices sur les booléens

1. Soient 5 variables booléennes A, B, C, D, E exprimant les propositions logiques suivantes :
A : Il faut prendre le chemin de droite
B : Il ne faut pas prendre le chemin de gauche
C : Il ne faut pas prendre le chemin du milieu
D : B dit la vérité et A est un menteur
E : C dit la vérité ou A est un menteur
Sachant que 3 des propositions sont fausses parmi les 5, quel chemin faut-il prendre ?
Réponse partielle : il faut essayer les 3 chemins et calculer le nombre de vrais et e faux.

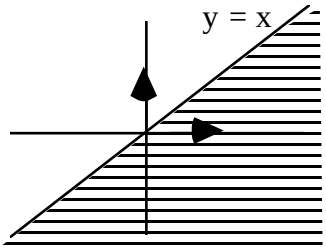
2. Soient 3 variables booléennes A, B, C exprimant les propositions logiques suivantes :
A : Le livre est à droite de la lampe
B : La lampe est à droite du crayon
C : Le crayon est à gauche du livre
En utilisant les opérateurs and, or et not appliqués à A, B, C, donnez l'équivalence booléenne des propositions suivantes :

1) Le crayon est l'objet le plus à droite.
2) La lampe est au milieu
3) Le livre n'est pas l'objet le plus à gauche
4) Si le crayon est au milieu, alors la lampe est à droite, sinon la lampe est au milieu.

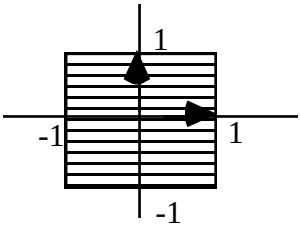
Réponse du 1) : not (B) and not(C) (peu importe la position du livre par rapport à la lampe)

3. Ecrire la suite d'instructions qui suppose que les coordonnées x,y d'un point P sont données par l'utilisateur et qui affecte la valeur true à une variable booléenne res si et seulement si P appartient à la zone hachurée suivante (donc false sinon) :

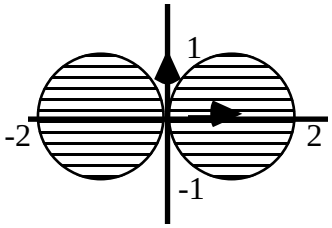
1)



2)



3)



2.3 Exercices sur les tableaux

1. Ecrire un programme qui demande à l'utilisateur 10 valeurs entières 1 par 1 dans un seul TEdit, qui les stocke au fur et à mesure dans un tableau et qui affiche dans un TMemo la plus grande valeur ainsi que son ordre d'insertion.

2. Ecrire un programme qui demande à l'utilisateur 10 valeurs entières correspondant à des notes entre 0 et 20 à l'aide d'1 seul TEdit, qui les stocke dans un TStringGrid et qui affiche dans un TLabel combien de notes sont supérieures ou égales à 10.

3. Soit T un tableau de 7 caractères correspondant à un mot. Ecrire les instructions permettant de modifier T de sorte que le mot soit renversé. Par exemple "bonjour" devient "ruojnob".

4. On désire effectuer des calculs sur les précipitations relevées chaque mois dans un même endroit.
a) Ajouter un TStringGrid pour que l'utilisateur puisse rentrer les valeurs des précipitations pour les 12 mois. Ecrire les instructions permettant d'afficher en première colonne de ce tableau le nom des 12 mois dès le lancement du programme.
b) Dans une procédure attachée à un bouton, écrire les instructions permettant d'afficher dans un TMemo l'écart de précipitation entre le mois le plus pluvieux et le mois le moins pluvieux ainsi que le nom de ces mois.
c) Ecrire les instructions permettant d'afficher l'histogramme des précipitations en utilisant l'instruction canvas.rectanble(x1, y1, x2, y2), de sorte qu'en abscisse on ait le numéro du mois et en ordonnée les précipitations, normalisées pour avoir entre 0 et 400 pixels, 400 pour le mois le plus pluvieux.

5. Un damier 10 x10 d'un jeu de dames se présente en début de partie de la façon suivante :

10		N		N		N		N		N
9	N		N		N		N		N	
8		N		N		N		N		N
7	N		N		N		N		N	
6										
5										
4		B		B		B		B		B
3	B		B		B		B		B	
2		B		B		B		B		B
1	B		B		B		B		B	
	1	2	3	4	5	6	7	8	9	10

a) Trouver une structure de données adaptée pour ce jeu.
b) Ecrire une procédure init qui place les pions dans le tableau comme indiqué sur la figure.

6. Jeu du démineur.
Un terrain rectangulaire miné est représenté par un tableau de NxM cases d'entiers. Au début du jeu, on placera un nombre X de mines de façon aléatoire dans le terrain : s'il n'y a pas de mine, la valeur de la case doit être égale à -2 et s'il y a une mine, la valeur de la case doit être égale à -1. Ensuite, le joueur doit donner les coordonnées x,y d'une case. S'il tombe sur une case avec -1, il a perdu, sinon, on affecte à la case x,y le nombre de mines présentes dans un carré 3 fois 3 centré en x,y. On affiche alors toutes les cases du terrain avec la convention suivante : si le nombre est négatif (-1 ou -2), on affiche un point d'interrogation, sinon on affiche la valeur de la case. Ensuite, on recommence, le joueur doit donner les coordonnées d'une autre case etc. Le jeu se termine lorsque le joueur tombe sur une mine ou lorsqu'il a trouvé toutes les cases non minées. Le nombre X de mines est demandé à l'utilisateur au début du jeu. N et M pourront être déclarés en constantes, toutes 2 égales à 5 pour commencer.

3. Fichiers textes génériques TInifile

3.1 Introduction

Les fichiers textes avec extension .ini sont exploités par Windows depuis les premières versions du système. Ils servaient notamment à initialiser (d'où leur extension) les différents programmes et l'environnement graphique de Windows. Ils sont définis dans Delphi par le type TInifile.

3.2 Organisation d'un fichier .ini

Un fichier texte de type TInifile est constitué de sections, placées entre crochets [et]. On peut définir autant de sections que nécessaires, les unes à la suite des autres.
Dans chaque section, on définit un nombre arbitraire de propriétés (définies en string), éventuellement 0, auxquelles on affecte une valeur au moyen du signe =. Retour à la ligne et lignes blanches sont considérés comme des séparateurs. Des commentaires peuvent être ajoutés après des ; .
Les fichiers .ini sont génériques dans le sens où le type des propriétés est quelconque. Il peut s'agir d'entiers, de réels, de booléens ou de string.
Exemple de contenu d'un fichier .ini :

```
[affichage]
largeur=800
hauteur=600
couleur=16M

[son]
présence enceintes=no
présence microphone=no

[police de caractères]
fonte=Times New Roman
taille=14
couleur=rouge
```

3.3 Utilisation d'un objet TInifile

TInifile est une classe, donc un type. On déclare un objet de type TInifile comme toutes les variables et on le crée à l'aide de la fonction create, avec comme paramètre un nom de fichier. Attention, par défaut, le répertoire est celui de Windows (utiliser GetDir pour obtenir le répertoire de l'application).

Déclaration en variable :	var f : TInifile;
Initialisation :	f := TInifile.Create (nom_de_fichier);
Fin d'utilisation :	f.free;

3.4 Propriétés et méthodes des objets TInifile

La propriété fondamentale des TInifile est Filename qui contient le nom complet du fichier texte. Les objets de type TInifile comportent essentiellement des méthodes (procédures et fonctions) dédiées à la récupération des données présentes dans un fichier .ini ou à l'écriture dans un tel fichier.

3.4.1 Lecture dans un fichier .ini

- function ReadString (Section, Ident, Default: String): String;

Récupère la chaîne de caractères (string) affectée à la propriété définie par "Ident" dans la section "Section". Si "Section" ou "Ident" n'existent pas, la fonction retourne la valeur du paramètre "Default".

- function ReadBool (Section, Ident: String; Default: Boolean): Boolean ;

Comme ReadString, mais pour récupérer une valeur booléenne.

- function ReadFloat (Section, Ident: String; Default: Double): Double ;

Comme ReadString, mais pour récupérer une valeur réelle en double précision.

- function ReadInteger (Section, Ident: String; Default: LongInt): LongInt ;

Comme ReadString, mais pour récupérer une valeur entière codée sur 4 octets.
Il existe aussi ReadDate, ReadTime et ReadDateTime pour récupérer des heures et des dates.

3.4.2 Ecriture dans un fichier .ini

Notez que l'appel à la méthode create ne génère automatiquement un nouveau fichier que si aucun fichier de même nom ne préexiste.

- procedure WriteString(Section, Ident, Value: String);

Affecte à la propriété définie par "Ident" dans la section "Section" la valeur "Value". Si "Section" ou "Ident" n'existent pas, elles sont créées automatiquement.

- procedure WriteBool(Section, Ident : String; Value: Boolean);

Comme WriteString, mais pour affecter un booléen.

- procedure WriteFloat(Section, Ident : String; Value: Double);

Comme WriteString, mais pour affecter un réel en double précision.

- procedure WriteInteger(Section, Ident : String; Value: LongInt);

Comme WriteString, mais pour affecter un entier.
Il existe aussi WriteDate, WriteTime, WriteDateTime

Pour la liste complète des méthodes, se référer à l'aide en ligne. On peut noter notamment les fonctions booléennes SectionExists pour tester l'existence d'une section et ValueExists pour tester l'existence d'une propriété dans une section donnée.

3.5 Exemple

Gestion d'un arbre généalogique. Les données concernant chaque personne sont hétérogènes et souvent incomplètes ou approximatives, ce qui compliquerait la gestion d'une base de données classique. Le TInifile facilite cette utilisation. Au démarrage de l'application, un fichier de type TInifile est lu afin de rentrer toutes les données => événement FormCreate

```
procedure TForm1.FormCreate(Sender: TObject);
var f : TInifile;
   section, sexe, repertoire, annee : string;
   i : integer;
   trouve : boolean;
   p : ppersonne;        // Structure de donnée de type record pour stocker les informations
begin                    // concernant une personne
  GetDir (0, repertoire) ;      // Permet d'obtenir le répertoire courant de l'application
  repertoire := repertoire + '\';        // Le fichier est placé en sous-répertoire
  f := TInifile.create(repertoire + 'genealogie.ini'); // Initialisalisation du TInifile
  i := 0;
  trouve := true;
```

```
//  Chargement des personnes
while trouve do
begin
  i := i + 1;                    // numéro de la personne
  section := IntToStr(i);
  if f.SectionExists(section)
  then begin
        p.identifiant := i;
        p.nom := f.ReadString(section,'nom', '');
        p.prenom := f.ReadString(section,'prenom', '');
        p.nom := f.ReadString(section,'nom', '');
        p.anneenaissance := f.ReadInteger(section,'anneenaissance', -1);
        p.moisnaissance := f.ReadString(section, 'moisnaissance', '');
        p.nomjeunefille := f.ReadString(section, 'nomjeunefille','');
        sexe := f.ReadString(section, 'sexe','m');
        p.sexe := sexe[1];
        p.journaissance := f.ReadInteger(section, 'journaissance',-1);
        p.anneedeces := f.ReadInteger(section,'anneedeces', -1);
        p.rem := f.ReadString(section, 'rem','');
        InsertInList(p);  // Procédure supposée écrite pour insérer une personne dans 1 liste
      end
  else trouve := false;
end;
        // Récupération des données concernant les parents :
        // ...etc.
 f.free;          // Libération de l'espace mémoire réservé à l'objet f
end;
```

Exemple du début d'un fichier "genealogie.ini"

```
[1]
nom=Dupond
prenom=Françoise
nomjeunefille=Durand
sexe=f
pere=2                    ; numéro du père
mere=15                   ; numéro de la mère
anneenaissance=1910
moisnaissance=janvier
journaissance=15
anneedeces=1978

[2]
nom=Dupond
prenom=Laurent
sexe=m
anneenaissance=1876
rem=disparu en mer en 1915 ? nombreuses données manquantes

[3]
nom=Dupond
prenom=Erik
sexe=m
anneedeces=1890
rem=orthographe prénom ? données incertaines fournies par Dupond Françoise
sexe=m
```

4. Introduction au langage objet

4.1 Utilité des objets

Le langage objet permet :

- L'encapsulation de champs et de méthodes dans une même classe, dans le but d'organiser et de structurer les données de manière cohérente.
- Une plus grande modularité, chaque classe pouvant souvent être construite indépendamment des autres.
- Une plus grande réutilisabilité, grâce à la définition de classes génériques, comme par exemple TStack, TQueue, TList, etc.
- L'héritage, c'est-à-dire la possibilité d'utiliser une classe préexistante et d'y rajouter des champs et des méthodes plus spécifiques.
- Le polymorphisme, c'est-à-dire la possibilité d'avoir des méthodes de même nom qui sont appelées de façon adaptative.

Toutes ces notions sont reprises et expliquées en plus de détail dans le cours.

4.2 Exemple

Les nouvelles classes se déclarent généralement dans la partie type de l'interface. Voici un exemple de création de classe.

```
Interface
type
        TClassPoint = class ( TObject )
          public
                x : real;
                y : real;
                constructor Create ( newx, newy : real);
                destructor Destroy;
                procedure RemiseAZero;
        end;
Implementation

constructor TClassPoint.Create ( newx, newy );
begin
        inherited create;       // Pour allouer la mémoire, on appelle le create hérité
        x := newx;
        y := newy;
end;

destructor TClassPOint.Destroy;
begin
        inherited Destroy;      // Rien à rajouter, on appelle le destructeur hérité
end;

procedure TClassPoint.RemiseAZero;
begin
        x := 0;
        y := 0;
end;
```

4.3 Classe TForm

La classe TForm est la classe qui définit les fiches. Lors de la définition d'une nouvelle fiche type, typiquement TForm1, on hérite des propriétés et méthodes de TForm grâce à la ligne :
TForm1 = class (TForm)
Lorsqu'on place de nouveaux composants, TLabel, TEdit, etc. sur la fiche, ceux-ci appartiennent à TForm1 mais pas à TForm.
Attention, TForm1 est une classe, définie dans la partie type. Pour créer une fiche de type TForm1, il faut déclarer une variable, par exemple Form1 qui est de type TForm1. Ceci est fait automatiquement par Delphi lors de la création d'une nouvelle fiche. Comme tous les objets, il faut attribuer un espace mémoire à form1, grâce à un appel à la méthode create ou éventuellement celle qui la remplace. Ceci est également réalisé de façon automatique par Delphi au niveau du fichier projet (le .dpr), grâce à la ligne :

```
Application.CreateForm(TForm1, Form1);
```

Tous les événements associés à des composants de la fiche, par exemple un clic sur un bouton, sont des méthodes de la fiche et sont donc déclarées dans TForm1.
On peut également ajouter à la fiche des propriétés et des méthodes qui ne sont pas associées à un composant de la fiche. Toutefois, l'intérêt du langage objet est de regrouper les propriétés et les méthodes de façon logique, il faut donc l'éviter.

4.4 Public / Private

Une propriété ou une méthode peut être déclarée dans la partie "public" ou "private" d'une classe. Si elle est dans la partie public, elle est accessible par un objet de n'importe quelle classe. Si elle est dans la partie private, seule une méthode de la même classe peut y faire référence, ou éventuellement du même fichier pascal. Par défaut, toutes les propriétés et méthodes sont publiques. Cependant, il faut au contraire privatiser au maximum pour garantir une séparation claire des objets.
Exemple :

```
Interface
type
        TClassObjet1 = class
                public  x : integer;
                private y : integer;
        end;
        TclassObjet2 = class
                private z : integer;
                procedure essai;
        end;
implementation
procedure TClassObjet2.essai;
var obj1 : TClassObjet1;
begin
        obj1 := TClassObjet1.Create;
        obj1.x := 3;            // Autorisé, car x est publique
        obj1.y := 4;            // Erreur à la compilation, car interdit, y est privé !
        z := 3;                 // Autorisé, car privé, mais appartient à la classe !
        ...
end;
```

4.5 Constructeur

constructor est une méthode spéciale de la classe appelée pour allouer de l'espace mémoire à l'objet et initialiser éventuellement certains champs. Généralement, on l'appelle "create", avec parfois des paramètres, en fonction des besoins.
Si on ne définit pas de constructeur, c'est la méthode Create de TObject qui joue ce rôle par défaut (car on hérite obligatoirement des méthodes de TObject). L'appel au constructeur doit se faire pour chaque objet créé. Si on cherche à accéder à la valeur d'un champ de l'objet alors qu'il n'y a pas eu d'appel au constructeur, il y a une erreur tout simplement parce qu'il n'y a pas d'espace mémoire valide attaché à l'objet.

- Exemple de construction :

```
var p : TClassPoint;
begin
        p := TClassPoint.Create ( 1, 2 );
```

Notez qu'après appel au constructeur, l'objet p est en fait une référence permettant à l'application de retrouver en mémoire tous les champs et méthodes de l'objet. Pour accéder à ces champs, il faut simplement ajouter un point et le nom du champ, par exemple p.x ou p.y.

4.6 Destructeur

destructor est une méthode spéciale de la classe appelée pour libérer l'espace mémoire de l'objet et éventuellement effectuer d'autres traitements. Généralement, on l'appelle "destroy", avec parfois des paramètres, en fonction des besoins. La méthode "free" est également souvent utilisée lorsqu'on désire seulement libérer l'espace mémoire.
Attention, après appel du destructeur ou de free, l'accès à un champ de l'objet entraîne une erreur, car il n'y a plus de référence valide en mémoire.

4.7 Affectation

Soit 2 objets p1 et p2 de la même classe. Si p1 a été créé avec appel du constructeur, alors
"p2 := p1" est autorisé et affecte la référence de p1 dans p2, de sorte que p1 et p2 référencent le même objet à la même position en mémoire.
Remarque : si p1 et p2 référencent 2 objets différents, l'affectation précédente conduit à perdre la référence de p2.
Si on veut que les champs de p1 soient copiées dans les champs de p2, il faut affecter chaque champ indépendamment.
Exemple :

```
p2.x := p1.x;
p2.y := p1.y;
```

Ainsi, p1 et p2 sont 2 objets différents (référence différente, position différente en mémoire), mais leurs champs ont la même valeur.
NB : Pour certaines classes prédéfinies de Delphi, il est possible d'affecter tous les champs d'un seul coup grâce à la méthode "Assign". De même, si on crée une nouvelle classe, on a parfois intérêt à définir une méthode pour copier les champs d'un objet et les affecter à un autre objet de la classe.

4.8 Héritage simple

Pour hériter des champs et des méthodes d'une classe, il suffit de préciser entre parenthèses lors de la déclaration la classe ancêtre. Exemples :

```
TFigure = class … end;
TRectangle = class ( TFigure ) … end;
```

TCarre = class (TRectangle) … end;
TEllipse = class (TFigure) … end;
Attention, il existe des règles contraignantes pour l'affectation si on manipule des objets de classes héritées.
Dans le cas des déclarations précédentes, si on a :
var F : TFigure; R : TRectangle; C : TCarre; E : TEllipse;
On a le droit d'écrire :
F := R; F := E; F := C; R := C;
Mais on n'a pas le droit d'écrire, entre autres :
R := F; E := F; C := F; C := R;

Explications :
F est un TFigure, donc à l'adresse mémoire référencée par F, on doit pouvoir accéder à tous les champs qui appartiennent au type TFigure. Si on fait F := R, on a effectivement à la nouvelle adresse de F tous les champs de la classe TFigure, puisqu'il s'agit d'un TRectangle qui hérite de TFigure. En revanche, si on fait R := F, on a à la nouvelle adresse de R tout ce qui concerne un TFigure, mais on ne peut avoir d'éventuels nouveaux champs appartenant spécifiquement à un TRectangle

Règle complémentaire :
Il est possible de donner à une méthode de la nouvelle classe le même nom qu'une méthode héritée. La méthode appelée en priorité est toujours celle qui correspond au type de la variable. Nous verrons dans les chapitres suivants d'autres règles concernant l'héritage et l'utilisation du même nom de méthode.

4.9 Exercice

Exercice 1 :
TClassPersonne est défini comme suit :

```
TClassPersonne = class
        nom : string;
        prenom : string;
        age : integer;
        information : string;
end;
```

Créez une nouvelle classe TListPersonnes qui hérite de TList et qui inclut les méthodes suivantes :

```
function Recherche ( nom, prenom : string ) : TClassPersonne; // Retourne le TClassPersonne
                                                               // recherché
procedure TriParAge;          // Trie la liste par ordre croissant en fonction de l'âge
```

Réponse :

```
interface
  type
        TListPersonnes = class ( TList )
                constructor create;
                function Recherche ( nom, prenom : string ) : TClassPersonne;
                procedure TriParAge;
                destructor destroy;
        end;
implementation
```

```
(************************************)
constructor TListPersonnes.Create;
begin
        inherited create;
end;
(************************************)
function TListPersonnes.Recherche( nom, prenom : string ) : TClassPersonne;
var     i : integer;      trouve : boolean;       p : TClassPersonne;
begin
        trouve := false;          i := 0;           p := nil;
        while not trouve and (i < count ) do
        begin
                p := items[i];   // Rappel, passage obligatoire car items[i] est de type pointeur
                if (p.nom = nom) and (p.prenom = prenom)
                then trouve := true
                else i := i + 1;
        end;
        Recherche := p;
end;
(************************************)
function compare ( p1, p2 : pointer ) : integer;
var     pers1, pers2 : TClassPersonne;
begin
        pers1 := p1;    pers2 := p2;    // Obligatoire pour avoir accès aux champs
        if pers1.age < pers2.age then compare := 1
        else if pers1.age = pers2.age then compare := 0
        else compare := -1
end;
procedure TListPersonnes.TriParAge;
begin
        sort ( compare );
end;
(************************************)
destructor TListPersonnes.Destroy;
begin
        inherited destroy;
end;
```

Commentaires : En héritant de TList, on bénéficie automatiquement des méthodes (par exemple count) et des champs de TList (notamment items). L'intérêt est de pouvoir manipuler un seul type pour gérer une liste de personnes. Par ailleurs, si TList était améliorée, on hériterait en même temps de ces améliorations, sans avoir à se soucier du détail des modifications apportées.

Exercice 2

Créer un objet TClassPoint pour manipuler des points en 3 dimensions, avec la possibilité de calculer la distance entre 2 points et une procédure d'affichage qui grossit le point de 1 à 10 pixels selon que la 3ème composante est proche ou loin de l'origine.

5. Héritage et polymorphisme

5.1 Surcharge

Au sein d'une même classe, on peut déclarer plusieurs méthodes avec le même nom, à condition qu'elles aient un nombre de paramètres ou/et un type différent. Il faut alors indiquer à la fin de la méthode le mot-clé overload (surcharge).
Exemple avec le constructeur :

```
TCercle = class
        X0, Y0 : real ;
        R : real ;
        // Construction à partir du centre et du rayon
        Constructor create ( x0, y0 : real ; R : real ) ;          Overload ;
        // Construction à partir du carré englobant
        Constructor create ( r : trect ) ;                         Overload ;
        // Construction à partir de 3 points
        Constructor create ( x1, y1, x2, y2, x3, y3 : real ) ;     Overload ;
        …
end ;
```

NB : S'il existe une méthode de même nom dans la classe ancêtre, 2 cas se présentent :

- Si la liste des paramètres est différente, elle peut aussi être appelée.
- Si la liste des paramètres est identique à une méthode redéclarée, elle est « masquée » et ne peut pas être appelée.

5.2 Méthode statique

Par défaut, toutes les méthodes sont dites « statiques ». Lorsqu'une méthode statique est appelée et qu'une méthode de même nom existe dans la classe ancêtre, c'est le type déclaré de l'objet déclaré en variable qui détermine quelle méthode est appelée.
Exemple :

```
Type
        TFigure = class
                Procedure dessin ;
                …
        end ;
        TRectangle = class ( TFigure )
                Procedure dessin ;
                …
        end ;
var
        Figure : TFigure ;
        Rectangle : TRectangle ;
Begin
        Figure := TFigure.create ;
        Figure.dessin ;                 // Appelle TFigure.dessin
        Figure.destroy ;
        Figure := TRectangle.create ;           // Autorisé, voir règles d'affectation
        Figure.dessin;                  // Appelle toujours TFigure.dessin!
        Figure.destroy ;
        Rectangle := TRectangle.create ;
        Rectangle.dessin ;              // Appelle TRectangle.dessin
        Rectangle.destroy ;
End ;
```

5.3 Méthodes virtuelles

Une méthode peut être déclarée virtuelle dans une classe ancêtre à l'aide du mot-clé « virtual ». Dans les classes qui héritent de la classe ancêtre, une méthode de même nom peut être déclarée surchargée avec le mot-clé « override ». La différence, c'est que cette fois-ci, la méthode appelée ne dépend pas du type déclaré de la variable, mais du type utilisé pour la construction de la variable.

Exemple :

```
Type
        TFigure = class
                Procedure dessin ;          virtual ;
                …
        end ;
        TRectangle = class ( TFigure )
                Procedure dessin ;          override ;
                …
        end ;
var
        Figure : TFigure ;
        Rectangle : TRectangle ;
Begin
        Figure := TFigure.create ;
        Figure.dessin ;                     // Appelle TFigure.dessin
        Figure.destroy ;
        Figure := TRectangle.create ;               // Autorisé, voir règles d'affectation
        Figure.dessin;                      // Appelle TRectangle.dessin !!!
        Figure.destroy ;
        Rectangle := TRectangle.create ;
        Rectangle.dessin ;                  // Appelle TRectangle.dessin
        Rectangle.destroy ;
End ;
```

En général, on utilise virtual pour pouvoir utiliser une méthode override définie dans une classe descendante, d'où le concept de virtualité.

NB1 : Si override est omis, c'est la méthode virtuelle qui est appelée.

NB2 : Une méthode peut également être déclarée « dynamic » au lieu de « virtual ». Le concept est pratiquement équivalent, l'idée étant que c'est à l'exécution, de façon dynamique donc, qu'il y a détermination de la méthode à appeler.

5.4 Méthode abstraite

Une méthode virtuelle ou dynamique est abstraite (« abstract ») si son implémentation n'existe pas. Le problème est résolu à l'exécution avec une méthode override dans une des classes descendantes.

Exemple :

```
Type
        TFigure = class
                Procedure dessin ;          virtual ; abstract ;
                …
        end ;
        TRectangle = class ( TFigure )
                Procedure dessin ;          override ;
                …
        end ;
var
        Figure : TFigure ;
Begin
        Figure := TFigure.create ;
        Figure.dessin ;                     // INTERDIT !!! car TFigure.dessin est abstraite
```

```
        Figure.destroy ;
        Figure := TRectangle.create ;          // Autorisé, voir règles d'affectation
        Figure.dessin;                  // Voilà le seul usage possible : appelle TRectangle.dessin !!!
        Figure.destroy ;
End ;
```

NB : Si une méthode héritée est abstraite, on ne peut pas utiliser « inherited » dans la méthode surchargée.

5.5 Self

« self » est un mot-clé faisant référence à l'objet lui-même dans une des méthodes de sa classe. On peut avoir besoin d'utiliser « self » pour plusieurs raisons, notamment si on veut passer l'objet en paramètre d'une méthode.

Exemple :

```
Interface
Type    TPeinture = class
                Procedure Affiche ;
                …
        end ;
        TFigure = class
                Procedure Draw ( peinture : Tpeinture ) ;
                …
        End;
implementation
procedure TPeinture.Affiche;
var f : TFigure ;
begin
        f := TFigure.create ;
        f.Draw ( self ) ;   // Self désigne ici l'instance de la classe TPeinture qui a permis d'appeler affiche
        f.destroy ;
end ;
…
var     p : TPeinture ;
begin
        p := TPeinture.create ;
        p.affiche ;                 // Ici, c'est p qui est l'instance de la classe TPeinture
        p.destroy ;
end ;
```

NB : self est comme une variable locale qui existe dans toutes les méthodes

5.6 Opérateurs de classe

- Transtypage

On peut modifier le type d'une variable à l'exécution en la précédant du nom d'un autre type puis de la variable entre parenthèses. Il faut éviter autant que possible cette action qui conduit souvent à des erreurs, mais dans certains cas il n'y a pas d'autres solutions.

Exemple :

```
Var   L : TList ;               p : TClassPoint ;
        p := L.items[0] ;
        x0 := p.x
```

Equivalent à `x0 := TClassPoint( L.items[0] ).x ;`

- Opérateur is

L'opérateur « is » permet de tester le type d'une variable objet.

NomObjet is NomClasse renvoie true si l'objet est de cette classe et false sinon.

Exemple :

```
If form1.activecontrol is TEdit then s := TEdit( form1.activecontrol ).text ;
```

6. Polymorphisme, étude d'un exemple

6.1 Problème

On dispose d'une liste de figures géométriques qui sont de type TClassCercle ou TClassRectangle qui héritent d'une classe TClassFigure. Pour gérer cette liste, on voudrait utiliser le type TList. Pour insérer une nouvelle figure dans la liste, on demande simplement à l'utilisateur soit les coordonnées du centre et la valeur du rayon, soit les coordonnées des 4 sommets du rectangle, puis on crée un objet de la classe appropriée et on l'ajoute à la liste.
Ensuite, on demande à l'utilisateur un nombre n et on affiche à l'écran la nième figure de la liste ainsi que le résultat du calcul de sa surface.
L'idéal serait de pouvoir écrire la procédure suivante :

```
var ListeFigures : TList;       // Variable globale
procedure TForm1.ButtonDemandeAffichageEtSurface ( sender : TObject );
var     n : integer;
        f : TClassFigure;       // Que ce soit un cercle ou un rectangle
begin
        n := StrToInt( editnum.text );          // editnum est un TEdit
        f := ListeFigures.Items[n-1];
        f.affiche ( canvas );
        label1.caption := 'Surface : ' + FloatToStr( f.surface );
end;
```

Une solution serait de mémoriser quelque part le fait qu'on manipule un cercle ou un rectangle, mais cette solution complique un peu les choses et n'est pas dans l'esprit de la programmation objet.

6.2 Solution

Il suffit de déclarer les classes de la façon suivante :

```
Interface       uses graphics, types ;          // Pour utiliser TCanvas et TPoint
type    TClassFigure = class // N'existe que pour bénéficier de l'héritage
                procedure affiche ( canvas : Tcanvas );     virtual;        abstract;
                function surface : real;                    virtual;        abstract;
        end;

        TClassCercle = class ( TClassFigure )
                x0, y0 : integer;
                R : integer;
                constructor create ( x, y : integer; rayon : integer );
                procedure affiche ( canvas : Tcanvas );     override;
                function surface : real;        override;
        end;

        TClassRectangle = class ( TClassFigure )
                p1, p2, p3, p4 : TPoint;
                constructor create ( a, b, c, d : TPoint );
                procedure affiche ( canvas : Tcanvas );     override;
                function surface : real;        override;
        end;
```

```
implementation
constructor TClassCercle.Create ( x, y : integer; rayon : integer );
begin
        inherited create;
        x0 := x;        y0 := y;        R := rayon;
end;

procedure TClassCercle.Affiche ( canvas : TCanvas );
begin
        canvas.ellipse ( x0-R, y0-R, x0+R, y0+R);
end;

function TClassCercle.Surface : real;
begin
        Surface := pi*R*R;
end;

constructor TClassRectangle.Create ( a, b, c, d : TPoint );
begin
        inherited create;
        p1 := a;        p2 := b;        p3 := c;        p4 := d;
end;

procedure TClassRectangle.Affiche ( canvas : TCanvas );
begin
        canvas.polyline ([ p1, p2, p3, p4, p1] );
end;

function TClassRectangle.Surface  : real;
begin
        Surface := sqrt(sqr(p2.x-p1.x)+sqr(p2.y-p1.y)) * sqrt(sqr(p4.x-p1.x)+sqr(p4.y-p1.y));
end;
```

Reste à écrire les procédures liées à l'interface.

```
procedure TForm1.ButtonCreeCercleClick ( sender : TObject );
var     C : TClassCercle;
begin
        C:= TClassCercle.Create ( StrToInt(edit1.text),
                                  StrToInt(edit2.text), StrToInt(edit3.text) );
        ListeFigures.Add ( C );
end;

procedure TForm1.ButtonCreeRectangleClick ( sender : TObject );
var R : TClassRectangle;      a, b, c, d : TPoint;
begin
        a.x := StrToInt( stringgrid1.cells[1,1] );      a.y := StrToInt( stringgrid1.cells[2,1] );
        b.x := StrToInt( stringgrid1.cells[1,2] );      b.y := StrToInt( stringgrid1.cells[2,2] );
        c.x := StrToInt( stringgrid1.cells[1,3] );      c.y := StrToInt( stringgrid1.cells[2,3] );
        d.x := StrToInt( stringgrid1.cells[1,4] );      d.y := StrToInt( stringgrid1.cells[2,4] );
```

```
        R := TClassRectangle.create ( a,b, c, d);
        ListeFigures.Add ( R );
end;
```

Le polymorphisme est ici mis en valeur grâce à l'abstraction des méthodes de Tfigure. Si on devait ajouter maintenant une classe TClassTriangle permettant de traiter également des figures de type triangle, il suffirait de rajouter les méthodes spécifiques à cette classe, d'ajouter une interface permettant de récupérer les paramètres du triangle et le reste du programme resterait identique.

6.3 Extension

Supposons maintenant que nous désirions pouvoir avoir le choix de dessiner en trait plein ou en pointillé, ainsi que la couleur du trait. Il s'agit ici de paramètres qui sont indépendants du type de figure. On peut donc rajouter des champs au niveau de la classe TFigure et du même coup implémenter un constructeur pour les initialiser.

```
TClassFigure = class // N'existe que pour bénéficier de l'héritage
                couleurdutrait : Tcolor;
                typedetrait : string;    // 'plein' ou 'pointillé'
                constructor create ( c : TColor; trait : string );
                procedure affiche ( canvas : Tcanvas );       virtual;          abstract;
                function surface : real;                      virtual;          abstract;
        end;

constructor TClassFigure.Create  ( c : TColor; trait : string );
begin
        inherited create;
        couleurdutrait := c;
        typedetrait := trait;
end;
```

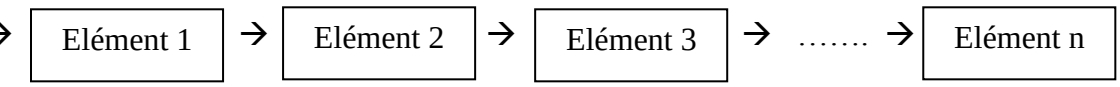
Il faudrait ensuite rajouter une interface pour que l'utilisateur puisse définir ses choix et modifier les constructeurs des classes descendantes afin que le "inherited create" comporte les 2 paramètres requis. Enfin, il faut modifier la procédure d'affichage comme suit :

```
procedure TForm1.ButtonDemandeAffichageEtSurface ( sender : TObject );
var     n : integer;
        f : TClassFigure;       // Que ce soit un cercle ou un rectangle
begin
        n := StrToInt( editnum.text );          // editnum est un TEdit
        f := ListeFigures.Items[n-1];
        canvas.pen.color := f.couleurdutrait;
        if f.typedetrait = 'plein' then canvas.pen.style := pssolid
        else canvas.pen.style := psdot;
        f.affiche ( canvas );
        label1.caption := 'Surface : ' + FloatToStr( f.surface );
end;
```

7. Listes chaînées

7.1 Représentation

Liste chaînée de n éléments :



- **Elément** : structure quelconque, par exemple un objet TImage ou une structure de type « record ».
- **Idée de chaînage** : à partir de l'élément courant, il est possible d'accéder à l'élément suivant.
- **Conséquence** : il suffit d'avoir l'accès au 1er élément pour pouvoir accéder à tous les éléments de la liste.

7.2 Gestion d'une liste chaînée

Pour gérer correctement une liste chaînée, il faut pouvoir :

- créer et supprimer la liste
- ajouter des éléments
- supprimer des éléments
- déplacer des éléments
- accéder aux éléments, notamment le 1er et le dernier
- connaître le nombre d'éléments
- chercher la présence d'un élément donné

7.3 Liste chaînée dans un tableau

On peut utiliser un tableau pour représenter une liste d'éléments.

Exemple :

```
// Définition d'un type pour représenter des personnes
type
Rpersonne = record
        Nom : string[50] ;
        Age : integer ;
        Profession : string[50] ;
End ;
// Définition d'un tableau de 100 personnes en variable
var T : array[1..100] of Rpersonne ;
```

Adresse mémoire	Valeurs
1000	Dupond 25 Architecte
1104	Durand 45 Ouvrier agricole
1208	Colin 25 Banquier
Etc.	

Avantages :
- Chaînage intégré dans la structure séquentielle du tableau : le suivant est placé en mémoire à l'adresse suivante.
- Parcours simple de la liste, à l'aide d'une variable représentant l'indice de l'élément.
- On peut accéder à n'importe quel élément à tout moment à partir de son indice.

Désavantages et limites :
- Réorganisation de la liste fastidieuse. Par exemple, comment insérer un nouvel élément au milieu ? Il faut opérer un décalage de tous les autres éléments. Même problème pour la suppression d'un élément, à moins de gérer des cases vides.
- On ne peut pas utiliser une liste d'éléments quelconques, par exemple un entier comme 1er élément, puis un record constitué de 2 string comme 2ème élément, etc. Cause : pour accéder à l'élément d'indice i, l'ordinateur calcule l'adresse du début du tableau et ajoute i fois la place occupée par l'élément structurant du tableau, qui ne peut donc varier.
- Si on utilise un tableau « statique », le nombre d'éléments doit être fixé à l'avance, même si on ne sait pas combien d'éléments il y aura dans la liste.

7.4 Tableaux dynamiques

Le problème dû aux tableaux statiques dont la taille est définie à l'avance peut être résolu par l'utilisation de tableaux dynamiques à l'aide de l'instruction « setlength ».
SetLength (nom_du_tableau, nombre_d_elements) ;
Exemple :
```
var t : array of RPersonne ;    // Pas d'indice défini à l'avance
puis dans le programme :
setlength ( t, 100 ) ; // Alloue de la place mémoire pour 100 éléments de type RPersonne
```

Remarques :
- Tous les tableaux dynamiques ont pour premier indice 0.
- Pour les tableaux multidimensionnels, il suffit de définir les tailles suivantes. Exemple : setlength (t, 100, 3) ; // Comme array[0..99, 0..2]

7.5 TList

TList est un type générique de Delphi pour gérer les listes chaînées à l'aide d'un tableau de pointeurs.

Type pointer :
« pointer » est un entier qui stocke une adresse en mémoire (ou pointeur) d'une variable. La manipulation de pointeurs est complexe, seul nous suffit pour l'instant la compréhension du concept.

Propriétés de TList :
- Count : integer ; // Nombre d'éléments de la liste
- Items : array of pointer ; // Tableau pointant sur les éléments
- Capacity : integer ; // Nombre maximal d'éléments que peut contenir la liste. Cette propriété n'a de sens que parce que TList exploite un tableau pour stocker les données.

NB : Les éléments sont rangés dans le tableau à partir de l'indice 0.

Méthodes de TList	Explications et commentaires
Constructor Create ;	Obligatoire pour initialiser une liste. Exemple. : list := TList.Create ;
Procedure Free ;	Libération de l'espace mémoire allouée à la liste
Function Add (element : pointer): Integer;	Ajoute un élément en fin de liste et renvoie son indice
Procedure Insert(Index: Integer; element : Pointer);	Insère un élément à la position spécifiée
Function Extract(element : pointer): Pointer;	Retire l'élément spécifié et renvoie son pointeur
Function Remove(element : Pointer): Integer;	Retire la première occurrence de l'élément spécifié et renvoie son indice
Procedure Delete(Index: Integer);	Retire l'élément dont l'index est spécifié
Function First: Pointer;	Comme Items[0], renvoie le 1er élément
Function Last : Pointer ;	Comme Items[list.count-1], renvoie le dernier élément
Procedure Move(CurIndex, NewIndex: Integer);	Déplace un élément d'un index vers un autre.
Function IndexOf(element : Pointer): Integer;	Renvoie l'indice de la première entrée du tableau Items contenant l'élément spécifié

7.6 Exemple

Voici un petit programme qui définit dans un premier temps une nouvelle classe d'objets nommée TClassPoint et qui exploite une liste chaînée pour insérer 2 points lors du click sur un premier bouton. Un deuxième bouton est utilisé pour récupérer le 1er et le dernier élément de la liste et afficher un petit cercle à la position des points.

```
INTERFACE
Type    // La fiche comporte un bouton Init et un bouton Affiche non décrits ici
        TClassPoint = class (TObject)
            x : integer;
            y : integer;
        end;

IMPLEMENTATION
var l : TList;
procedure TForm1.ButtonInitClick(Sender: TObject);
var   p1, p2 : TClassPoint;
begin
  l := TList.Create;
  p1 := TClassPoint.create;
  p2 := TClassPoint.Create;
  p1.x := 1;
  p1.y := 100;
  p2.x := 50;
  p2.y := 50;
  l.Add( p1);
```

```
  l.Add( p2);
end;

procedure TForm1.Button2Click(Sender: TObject);
var p : TClassPoint;
begin
  p := l.first;
  canvas.ellipse (p.X, p.Y, p.X+2, p.y+2);
  p := l.last;
  canvas.ellipse (p.X, p.Y, p.X+2, p.y+2);
end;
```

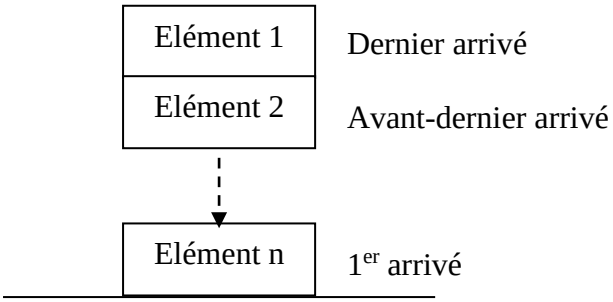
7.7 Exercices

Définissez une classe TClassPoint. A l'aide d'un bouton, initialisez une liste de 30 points avec des coordonnées aléatoires entre 100 et 300.
Ajoutez un 2ème bouton pour afficher tous les points de la liste sous la forme d'un cercle rouge (bord et intérieur) de 5 pixels de diamètre centré sur le point (méthode canvas.ellipse)
Ajoutez un 3ème bouton pour dessiner en bleu les 3 premiers points de la liste dont les coordonnées sont inférieures à 250.

8. Piles

8.1 Représentation

Dans le jargon informatique, une pile est une liste chaînée d'éléments dont la gestion s'apparente à celle d'une « pile » (stack en Anglais) classique, par exemple une pile de dossiers sur un bureau. Les éléments sont donc représentés les uns au-dessus des autres, le premier inséré en bas et le dernier en haut.



8.2 Gestion d'une pile

- L'insertion d'un nouvel élément se fait toujours en haut de la pile.
- Pour la consultation des éléments de la pile, on accède toujours à l'élément en haut d'abord.
- Pour la suppression, de même, on supprime toujours celui du haut d'abord.

Ainsi, c'est toujours le dernier élément arrivé qui est consulté et supprimé en premier. On parle de **gestion LIFO** (Last In First Out).

8.3 Exemples de piles

Ctrl Z :

Dans un éditeur de texte et dans bien d'autres programmes, il est possible d'annuler les dernières commandes et de revenir en arrière. C'est d'abord la dernière commande qui est annulée, puis l'avant-dernière etc. Les commandes ont été empilées au fur et à mesure du travail dans une pile. Lors de l'annulation, c'est la dernière commande enregistrée qui est annulée et qui est sortie de la pile, on a bien une gestion LIFO.

Pile pour la gestion des appels de fonction :

A chaque appel de procédure ou de fonction, le programme réserve de la place en mémoire pour les valeurs des paramètres et des variables locales. Il gère cette réservation de place comme une pile : si une nouvelle procédure ou fonction est appelée, il réserve de la place en haut de la pile et si une procédure ou fonction se termine, il libère la place occupée, il dépile.

Remarque : il arrive que le programme n'ait pas assez de place en mémoire pour un nouvel appel de procédure ou de fonction. Il est possible de modifier la taille mémoire maximale de la pile dans le menu Projet, Options et éditions des liens.

8.4 Objets TStack

TStack est une classe d'objets dédiés à la gestion des piles. L'élément de la pile peut être n'importe quel objet ou pointeur.

TStack n'a pas de propriété mais uniquement des méthodes.

TStack est défini dans l'unité Contnrs, il faut donc ajouter "uses contnrs" dans la partie uses de l'unité.

8.5 Méthodes de TStack

constructor Create;	Comme toutes les classes, il y a un constructeur
function Count : integer;	Renvoie le nombre d'éléments de la pile
procedure Push (p : Pointer);	Place l'élément p en haut de la pile = empile
function Peek : pointer;	Récupère l'élément en haut de la pile sans dépiler
function Pop : pointer;	Récupère l'élément en haut de la pile et le supprime de la pile = dépile
function AtLeast (n : integer) : boolean;	Détermine s'il reste au moins n éléments dans la pile et retourne true ou false.

8.6 Exercice

Développez une application permettant de dessiner des points noir n'importe où sur un composant TImage contenant une image quelconque. Utilisez pour cela OnMouseDown du composant TImage.
Ajoutez un bouton "Annuler" qui annule le dernier point dessiné et rétablit la couleur initiale. En gérant la liste des points dessinés avec une pile, il doit être possible d'annuler le nombre de points désirés en cliquant autant de fois que nécessaire sur le bouton "Annuler".

Solution :

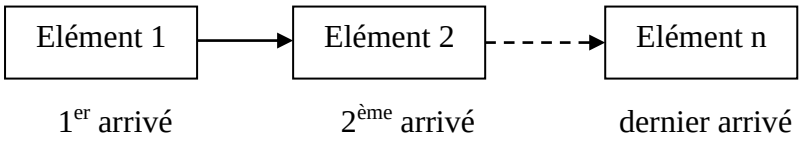
```
// Déclaration de TForm1 non décrite ici
type
TClassPoint = class
      x,y : integer;
      couleur : TColor;
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}

uses Contnrs;
var  pile : TStack;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var p : TClassPoint;
begin
  // Mémorisation de la position du point et de sa couleur initiale
  p := TCLassPoint.Create;
  p.x := x;
  p.y := y;
  p.couleur :=  image1.Picture.Bitmap.Canvas.Pixels[x,y];
  // On empile
  pile.Push(p);
  // On met du noir à la place
  image1.Picture.Bitmap.Canvas.Pixels[x,y] := 0;
end;
```

```
procedure TForm1.ButtonAnnulerClick(Sender: TObject);
var p : TClassPoint;
begin
   // Si aucun point dans la pile, il n'y a rien à annuler
  if pile.Count = 0 then exit;
  // Récupération du dernier point dessiné et suppression de la pile
   p := pile.pop;
   // On remet la couleur initiale
   image1.Picture.Bitmap.Canvas.Pixels[p.x,p.y] := p.couleur;
   // libération de la place mémoire du point, devenu inutile
   p.free;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  // Fin du programme, libération de l'espace mémoire réservé à la pile
  pile.free;
end;

initialization
   pile := TStack.Create;

end.
```

9. Files

9.1 Représentation

Dans le jargon informatique, une file est une liste chaînée d'éléments dont la gestion s'apparente à celle d'une « file d'attente » (queue en Anglais) classique, par exemple une file de voitures attendant à un feu rouge pour le passage d'un carrefour. Les éléments sont donc représentés les uns derrière les autres, le premier arrivé restant le premier de la liste.



9.2 Gestion d'une file

- L'insertion d'un nouvel élément se fait toujours en fin de liste, "à la queue comme tout le monde !".
- Pour la consultation des éléments de la file, on accède toujours à l'élément arrivé en 1er.
- Pour la suppression, de même, on supprime toujours celui arrivé en premier.

Ainsi, c'est toujours le premier élément arrivé qui est consulté et supprimé en premier. On parle de **gestion FIFO** (First In First Out).

9.3 Exemple de file

Gestion des tâches d'impression.

L'impression d'un document est généralement assez longue, en comparaison de la durée d'envoi des données à imprimer. Si un autre ordre d'impression est donné, celui-ci est placé dans une liste d'attente. Le traitement de la liste des documents à imprimer se fait en fonction de l'ordre d'arrivée. La gestion de la liste est donc de type FIFO, premier arrivé, premier traité.

Simulation du passage des voitures à un croisement

Imaginons un croisement routier avec des feux pour gérer le passage des voitures. Toutes les voitures qui vont tout droit se placent dans la même file, de même pour ceux qui tournent à gauche et ceux qui tournent à droite. Lorsque le feu est rouge, la file d'attente grandit au fur et à mesure que les voitures arrivent. Lorsque le feu passe au vert, les voitures repartent selon leur ordre d'arrivée. Là encore, si on simulait leur passage, on utiliserait une liste chaînée de type "file", avec une gestion FIFO, premier arrivé, premier à repartir.

De manière générale, les files d'attente sont typiquement utilisées lorsqu'il y a un processus de gestion asynchrone avec une exécution relativement lente du traitement principal et une arrivée aléatoire ou par petits groupes des données à traiter. Ces données sont donc placées dans une liste d'attente, en vue d'être traitées selon leur ordre d'arrivée.

Autre exemple : une boîte vocale enregistrant les messages. Si le message n'est pas supprimé, on peut supposer qu'il y a création d'une nouvelle file au fur et à mesure que la première est traitée. Lors de la consultation, l'accès à un message nécessite d'écouter tous ceux qui précèdent, ce qui correspond bien au traitement FIFO.

9.4 Objets TQueue

Il existe une classe "TQueue" pour gérer les files.
Comme pour TStack (piles), TQueue ne possède aucune propriété, uniquement des méthodes.
TQueue est défini dans l'unité Contnrs, il faut donc ajouter "uses contnrs" dans la partie uses de l'unité.

9.5 Méthodes de TQueue

constructor Create;	Comme toutes les classes, il y a un constructeur
function Count : integer;	Renvoie le nombre d'éléments de la file
procedure Push (p : Pointer);	Place l'élément p à la fin de la file.
function Peek : pointer;	Récupère l'élément en début de file sans le supprimer.
function Pop : pointer;	Récupère l'élément en début de file et le supprime de la file.
function AtLeast (n : integer) : boolean;	Détermine s'il reste au moins n éléments dans la file et retourne true ou false.

9.6 Exercice

Simulation de la gestion d'une imprimante.
Définissez une classe TClassDocument avec les propriétés suivantes :
nom du document de type string, taille en octets de type integer et état de type string ("en cours" ou "en attente").
Ajoutez un TTimer dont la propriété enabled sera définie à true lorsqu'une impression est démarrée et à false lorsque l'impression est terminée, pour simuler le temps d'attente.
A l'aide de plusieurs TEdit et d'un bouton "Impression", déclenchez un événement d'impression virtuelle en créant un TClassDocument avec les paramètres écrits dans les TEdit, placez le dans une file d'attente et appelez une procédure "LancementImpression" si le TTimer a sa propriété enabled à false (imprimante inactive).
Traitement de la procédure "LancementImpression" : on récupère le 1er document à traiter dans la liste. On change l'état, on met "en cours" et on active un TTimer avec une attente proportionnelle au nombre d'octets de la taille du document.
Traitement de l'événement du TTimer : on supprime le 1er document de la liste et si la liste n'est pas vide, on relance une impression en appelant "LancementImpression".
Ajoutez un TStringGrid pour visualiser les documents en cours ou en attente d'impression : Ajout d'un élément lors du clic sur le bouton et remontée d'un cran la liste des documents lors de la fin de l'impression d'un document (donc dans l'événement du TTimer).

Solution :

```
// Procédures de TForm1 non décrites ici
type
  TClassDocument = class (TObject)
     nom : string;
     taille : integer;
     etat : string;
  end;

var
  Form1: TForm1;
implementation
{$R *.dfm}
```

```
uses contnrs;

var f : TQueue;

procedure TForm1.FormCreate(Sender: TObject);
begin
  StringGrid1.Cells[0,0] := 'Nom du document';
  StringGrid1.Cells[1,0] := 'Taille du document';
  StringGrid1.Cells[2,0] := 'Etat de l''impression';
  f := TQueue.Create;
end;

procedure lancementimpression;
var doc : TClassDocument;
begin
      doc := f.peek;
      Form1.Timer1.Interval := doc.taille *1000;
      Form1.Timer1.Enabled := true;
      doc.etat := 'en cours';
      form1.stringgrid1.cells[2,1] := doc.etat;
end;

procedure TForm1.Button1Click(Sender: TObject);
var doc : TClassDocument;
begin
 doc := TClassDocument.Create;
 doc.nom := edit1.Text;
 doc.taille := StrToInt(edit2.text);
 doc.etat := 'en attente';
 f.Push(doc);
 // Affichage dans StringGrid
 stringgrid1.cells[0,f.Count] := doc.nom;
 stringgrid1.Cells[1,f.count] := IntToStr(doc.taille);
 stringgrid1.Cells[2,f.count] := doc.etat;
 // Lancement de l'imprimante si pas déjà fait
 if timer1.Enabled = false  then lancementimpression;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var i : integer;
  doc : TClassDocument ;
begin
 // Impression terminée
 Form1.Timer1.Enabled := false;
 // suppression affichage
 for i:=1 to f.count do
 begin
        form1.stringgrid1.cells[0,i] := form1.stringgrid1.cells[0,i+1];
        form1.stringgrid1.Cells[1,i] := form1.stringgrid1.cells[1,i+1];
        form1.stringgrid1.Cells[2,i] := form1.stringgrid1.cells[2,i+1];
 end;
 // Suppression de la file d'attente
 doc := f.Pop;
 doc.free ;
 // Lancement de l'impression suivante
 if f.Count > 0  then lancementimpression;
end;
end.
```

10. Listes ordonnées

10.1 Concepts

Une liste ordonnée est une liste d'éléments sur lesquels est appliquée une relation d'ordre pour ordonner / trier les éléments.
Soient X,Y deux éléments quelconques de la liste, et < une relation d'ordre, on doit avoir :
X < Y ou Y < X, ou éventuellement X et Y au même niveau.
Pour gérer une liste ordonnée, on utilise de façon classique l'objet TList. Ses éléments étant stockés dans un tableau, on peut accéder directement à un élément en fonction de sa position (ou "index") dans le tableau.
Remarque : si on veut accéder à un élément en fonction de son rang, ou si on veut trier les éléments en fonction d'une autre relation d'ordre, il faut pouvoir accéder à n'importe quelle position dans la liste, pas seulement la première ou la dernière, on ne peut donc pas utiliser de pile ou de file.

10.2 Tri par insertion

Il existe un grand nombre d'algorithmes pour le tri. Trier une liste est en général difficile, un exemple simple de tri est proposé ici, le tri par insertion. L'objectif n'est pas de retenir cet algorithme, car nous verrons un peu plus loin qu'il existe dans TList une méthode qui effectue un tri de façon automatique.

```
const n=100;
type tab = array[1..n] of real;
procedure tri ( var t : tab );
var     i,j, indicedumin : integer;
        min, aux : real;
begin
        // On va chercher le plus petit, le placer en 1, puis le 2ème plus petit etc.
        for i:=1 to n-1 do
        begin
                // Recherche du ième minimum
                min := t[i];
                indicedumin := i;
                for j:=i+1 to n do
                        if (t[j] < min)
                        then begin
                                min := t[j];
                                indicedumin := j;
                             end;
                // Placement du min en ième position par permutation avec le ième actuel
                aux := t[i];
                t[i] := min;
                t[indicedumin] := aux;
        end;
end;
```

10.3 Tri avec TList

Méthode de TList effectuant un tri : procedure Tlist.Sort(Compare: TListSortCompare);
avec un nouveau type prédéfini :

type TListSortCompare = function (p1, p2: Pointer): Integer;

cette fonction retourne 1 si l'objet p1 doit être classé avant p2, 0 si p1 et p2 sont au même rang et -1 si p2 doit être classé avant p1.

Explications : il est possible de passer en paramètre d'une procédure ou d'une fonction le nom d'une autre procédure ou fonction. En l'occurrence, TList.Sort utilise une fonction de comparaison de type TListSortCompare. Pour qu'une fonction soit de ce type, il suffit que sa déclaration soit identique, c'est-à-dire le même nombre de paramètres et un integer comme valeur de retour.

10.4 Exemple

Exemple :

Soit TClassPoint le type défini de la façon suivante :

```
TClassPoint = class (TObject)
        x, y : integer;
end;
// Déclaration de la fonction de comparaison.
// Choix : on ordonne les points définis par p1 et p2 en fonction de leur ordonnée.
function MaFonctioDeComparaison ( p1 , p2 : pointer ) : integer;
var obj1, obj2 : TClassPoint;
begin
        obj1 := p1;     obj2 := p2;
        if obj1.y > obj2.y then MaFonctioDeComparaison := 1
        else if obj1.y = obj2.y then MaFonctioDeComparaison := 0
        else MaFonctioDeComparaison := -1;
end;
```

// Ailleurs dans le programme, si l est de type tlist, on peut faire un tri de la manière suivante :
l.sort (MaFonctioDeComparaison);

Remarque :

Si on veut trier l différemment, il suffit de définir d'autres fonctions de comparaison et de passer en paramètre la fonction de comparaison souhaitée.

10.5 Exercice

Développez une application permettant de placer dans une liste des objets de type TClassPersonne, comprenant les propriétés "nom" de type string et "age" de type integer, à partir des données entrées au clavier par l'utilisateur. La liste des personnes pourra être visualisée dans un TMemo. Ajoutez ensuite 2 boutons, le premier pour effectuer un tri en fonction du nom, le deuxième pour effectuer un tri en fonction de l'âge, suivi de l'affichage de la liste triée.

Réponse :

// La définition de TForm1, relativement évidente, n'est pas affichée.

```
 TClassPersonne = class
      nom : string;
      age : integer;
 end;

var       Form1: TForm1;
```

```
implementation
        {$R *.dfm}
var l : TList;

function comparenom ( p1, p2 : pointer ) : integer;    // comparenom est de type TListSortCompare
var x, y : TClassPersonne;
begin
  x := p1;          y := p2;
  if x.nom > y.nom then comparenom := 1
  else if x.nom = y.nom then comparenom := 0
  else comparenom  := -1;
end;

function compareage ( p1, p2 : pointer ) : integer;     // compareage est de type TListSortCompare
var x, y : TClassPersonne;
begin
  x := p1;          y := p2;
  if x.age > y.age then compareage := 1
  else if x.nom = y.nom then compareage := 0
  else compareage  := -1;
end;

procedure TForm1.affiche;          // Affiche la liste des personnes, nom et âge, dans le TMemo
var     p : TClassPersonne;
        i : integer;
begin
  memo1.Clear;
  for i:=0 to l.Count-1 do
  begin
     p := l.items[i];
     memo1.Lines.Add(p.nom + ' ' + IntToStr(p.age));
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var p : TClasspersonne;
begin
  p := TClassPersonne.Create;
  p.nom := edit1.Text;
  p.age := StrToInt(edit2.text);
  l.Add(p);
  affiche;
end;

procedure TForm1.Button2Click(Sender: TObject);  // Tri par nom
begin
 l.Sort(comparenom);
 affiche;
end;

procedure TForm1.Button3Click(Sender: TObject);  // Tri par âge
begin
  l.Sort(compareage);
  affiche;
end;

initialization
 l := TList.Create;
end.
```

11. Images

11.1 Structures de données

TImage :

- width
- height
- bitmap
- bitmap.canvas.pixels[x,y] : tableau bidimensionnel de TColor.

Pour se ramener à R,V,B le mieux est de définir un tableau tridimensionnel de byte. Pour obtenir R, V, B à partir d'un codage sur 32 bits (TColor), il faut récupérer chaque octet. Le plus rapide est d'effectuer des décalages à gauche ou à droite pour se débarrasser des octets non désirés à l'aide shl et shr (shift left et shift right).

```
for x:=1 to im1.width do
   for y:=1 to im1.height do
   begin
       colorimage := im1.bitmap.canvas.pixels[x-1,y-1];
        t1[x,y,1] := (colorimage shl 24) shr 24;
        t1[x,y,2] := (colorimage shl 16) shr 24;
        t1[x,y,3] := (colorimage shl 8) shr 24;
   end;
```

Si on ne connaît pas la taille de l'image a priori, il faut utiliser un tableau dynamique et l'initialiser grâce à setlength(dimensionx, dimensiony).

11.2 Algorithmes sur les images

- Sélection d'une couleur
- Transformation en niveaux de gris
- Inversion
- Ajouter du flou
- Rehaussement du contraste

12. Révision, synthèse

12.1 Exercices de synthèse

Exercice 1
On considère la déclaration suivante :
```
const N = 100;
type tab = array[1..N] of real;
```
Ecrire une fonction qui prend en paramètres un tableau t de type tab et un seuil s de type real et qui calcule le nombre d'éléments de t strictement supérieurs à s.

Exercice 2
On considère les déclarations suivantes :
```
type
TClassMessage = class
        message : string;
        heuredarrivee : integer;        // exprimée en secondes
        priorite : integer;       // entre 0 (faible) et 5 (haute)
end;
var     L : TList;       // Liste d'objets de type TClassMessage
        F : TQueue;    // File d'attente d'objets de type TClassMessage
```

2.1 Ecrire une procédure qui prend en paramètres un message (de type string), une heure d'arrivée et une priorité et qui insère dans F un TClassMessage formé par ces données.

2.2 Ecrire une procédure qui trie la liste L en fonction de la priorité (priorité 5 en premier), et, à priorité égale, de l'heure d'arrivée (heure d'arrivée la plus petite en premier).

2.3 Ecrire une procédure qui crée l'objet F, appelle la procédure de tri du 2.2 et insère dans F tous les éléments de L.

2.4 Ecrire une procédure qui prend en paramètre un nombre n et qui supprime de F et de L les n premiers éléments, en prenant en compte la possibilité que les listes soient vides.

12.2 Bilan

Ce cours a permis d'introduire de nombreuses notions comme les piles, les files, les listes ordonnées, les fichiers textes génériques (TInifile), l'utilisation d'objets et de renforcer les acquis éventuels des années précédentes en résumant les principales caractéristiques de la programmation Pascal Delphi.
Au niveau des travaux pratiques, on peut noter la part importante prise par la conception de l'interface, la gestion de l'affichage et l'interaction avec l'utilisateur. En sciences cognitives, on le justifie en disant qu'il s'agit de placer la problématique liée à l'interface et aux facteurs humains, souvent négligés, au même niveau que la problématique algorithmique.
Les travaux pratiques ont également permis d'aborder la problématique de la simulation dans le domaine des sciences humaines et sociales. C'est là qu'il faut trouver une partie de la justification de l'apprentissage d'une plateforme de développement d'applications informatiques. Il existe en effet de nombreux problèmes qui ne trouvent pas de solution mathématique simple. Il faut alors avoir recours aux simulations pour tester un modèle et

faire de la prospective. Notons à ce sujet l'existence en 2003 d'un appel d'offres CNRS pour financer des projets qui rentrent dans le cadre des "Systèmes Complexes en Sciences Humaines et Sociales" (voir : http://www.recherche.gouv.fr/recherche/fns/scshs.htm).
Exemples de projets financés :

- Emergence et évolution des systèmes urbains : un modèle de simulation en fonction des conditions historiques des interactions spatiales.
- ELICCIR, Emergences, de l'Individu au Comportement Collectif : Interactions et Représentations. (systèmes multi-agents pour tester la prise de décision locale ou collective concernant le marché du travail et les échanges économiques).
- Dynamique spatio-temporelle des épidémies et réseaux stochastiques multi-échelles.

ANNEXES : LISTE DES TD-TP

1. Petits exercices de rappel

2. Utilisation du Canvas

3. Utilisation du Timer

4. Canvas + ttimer

5. je sais pas quoi

6. TInifile

7.

8. Piles, TStack

9. Files, TClassList

10. TListbox

11. Listes chaînées

12.

Licence MASS / SciCo
3ème année

TD-TP 1 et 2: Exercices de rappel Pascal / Delphi

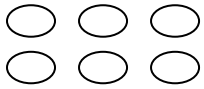
Les exercices pourront être choisis en fonction du niveau des étudiants.

1. Utilisation du if then else

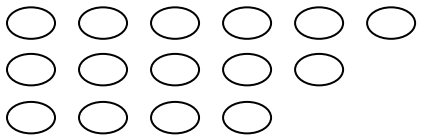
Ecrire un programme pour jouer à "Devine le nombre". L'ordinateur choisit un nombre de façon aléatoire entre 0 et 9999 et l'utilisateur tente de le deviner. Pour cela, il tape un nombre et l'ordinateur répond trop grand, trop petit ou gagné. Le nombre de coups doit être affiché. A vous de concevoir l'interface pour que le jeu soit convivial.
Rappel : pour générer un nombre aléatoire entre 0 et x, il faut faire un appel à la fonction random(x). Pour que le générateur soit initialisé, il faut également appeler randomize, 1 seule fois en début de programme.

2. Utilisation du for

canvas.ellipse (x1,y1, x2, y2) permet d'afficher une ellipse inscrite dans le rectangle définit par les coins opposés (x1, y1) et (x2,y2). Ecrire un programme qui demande 2 entiers x et y à l'utilisateur et qui affiche les dessins suivants (à vous de concevoir l'interface appropriée) :
2.1 x rangées de y cercles de diamètre 10 pixels et espacés de 10 pixels.
Exemple x = 2 e y =3 :



2.2 Idem, avec 1 cercle de moins sur la ligne d'en dessous
Exemple x = 3, y = 6



2.3 Un triangle rectangle et isocèle de ronds, c'est-à-dire x rangées avec x ronds en 1ère ligne, puis x-1 en 2ème ligne etc. jusqu'à 1 rond en dernière ligne (y n'est pas utilisé).

2.4 Exercice difficile à faire en fin de séance : un carré de ronds (comme pour 2.1 avec x=y), sauf qu'à l'intérieur du carré, il y a un vide qui a une forme circulaire de diamètre égal à x petits ronds. Le dessin n'est intéressant que pour de grandes valeurs de x.

3. Utilisation du while
Reprendre le 2.1, le 2.2 et le 2.3 en évitant d'utiliser la boucle for.

4. Ecrire un programme qui demande à l'utilisateur un entier n et qui calcule n!.
4.1 Avec un for
4.2 Avec un while
4.3 En utilisant la récursivité

5. Créer une application avec les caractéristiques suivantes :
- une fiche standard de type TForm qui prend tout l'écran
- un label dans lequel est écrit 'Evident mon cher Watson'
- 3 boutons permettant de changer la police du texte du label en times new roman, courier ou arial.
- 1 boîte à cocher TCheckbox permettant de rendre le label visible si elle est cochée et invisible si elle est décochée.
- 1 boîte d'édition TEdit et 1 bouton permettant d'ajuster la taille de la police du label en fonction de la valeur rentrée dans le TEdit.
- 1 boîte d'édition TEdit et 1 bouton permettant de modifier la position verticale du label sur la fiche en fonction de la valeur rentrée dans le TEdit.
- 1 bouton permettant de rendre le bouton précédent inactif (bien que toujours visible).

6. Créer une application avec les caractéristiques suivantes :
- une fiche standard de type TForm qui prend tout l'écran et de couleur bleue
- un label sur lequel il y a écrit 'veuillez patienter 3 minutes'
- un bouton jaune sur lequel il y a écrit en rouge 'annulation attente'
- si l'utilisateur déplace la souris au-dessus du bouton (avant même de cliquer), ce dernier se positionne aussitôt au-dessus de la souris, comme si le bouton voulait échapper à l'utilisateur … Si le bouton arrive en haut de la fiche, déplacer le bouton tout en bas de la fiche. (Utiliser l'événement OnMouseMove).