Initation Delphi — TP

Christopher “Vjeux” Chedeau | chedea_c

24 Octobre 2008

Table des matières

Chapitre 1

Introduction

1.1 Environnement de Travail

- Lancer Delphi 2007
- Fichier / Nouveau / Autres / Projets Delphi / Application console
- Supprimer la portion de code suivante

```
try
{ TODO -oUser -cConsole Main : Insert code here }
except
  on E:Exception do
    Writeln(E.Classname, ': ', E.Message);
end;
```

- Tout Enregistrer (Ctrl + Shift + S) en 'TPDelphi.dproj'
- Dans le panel de droite : Click droit sur 'TPDelphi.exe' / Ajouter Nouveau / Unité
- Enregistrer (Ctrl + S) en 'u_facto.pas'

Vous venez de mettre en place votre environnement de travail. Il est actuellement constitue de deux fichiers :

- Fichier 'TPDelphi.dpr'

```
program TPDelphi;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  u_facto in 'u_facto.pas;

begin

end.
```

– Fichier ‘u_facto.pas‘

```
unit u_facto;

interface

implementation

end.
```

★ **Remarque** On ne peut pas nommer un fichier et une fonction ou un type avec le même identifiant. Ainsi il est pratique de placer un préfixe différent devant les identifiants. Par exemple `u_` devant les noms de fichiers, `t_` devant les noms de types ...

★ **Attention** Il est important de bien séparer deux choses fondamentales. Les fichiers .pas sont des unités qui contiennent des fonctions avec du code. C'est là où seront situés les algorithmes. Pour tester ces algorithmes, il est vivement conseillé de le faire dans le .dpr ou même mieux dans une autre unité et de placer uniquement un appel dans le dpr.

1.2 Hello World !

Comme tout bon tutorial qui se respecte, la première chose à vous faire faire est d'afficher `Hello World!`. Pour cela il vous faudra utiliser la procédure `WriteLn` (comme ‘Write Line‘). Tous les paramètres donnés seront convertis en chaîne de caractères (pour les types prédéfinis), concaténés puis affichés sur la sortie standard.

Dans un premier temps nous allons le placer dans le fichier `TPDelphi.dpr`.

```
begin

  WriteLn('Hello World!');
  ReadLn;

end.
```

★ **Remarque** La fonction `ReadLn` va attendre que l'utilisateur appuye sur Entrée. Cela permet que le programme ne s'arrête pas directement mais nous laisse le temps de lire notre `Hello World!`

1.3 Première Fonction

Votre première fonction va écrire `Hello World!` sur la sortie. Vous devrez la placer dans le fichier `u_facto.pas`. Celui-ci est constitué de deux parties. L'`implementation` contient toutes les définitions (ou prototypes) des fonctions, quant à l'`implementation` qui contient leur code. Voici le prototype de la fonction à réaliser :

Prototype `procedure HelloWorld();`

★ **Remarque** En Delphi on distingue deux types de fonctions, les `procedure` et les `function`. La seule différence est que la `procedure` ne renvoie pas de valeur alors que la `function` oui.

Il vous faudra donc écrire votre prototype dans la section `interface` de l'unité et écrire le code de la fonction dans la partie `implementation`. Voici à quoi doit ressembler le fichier `u_facto.pas` :

```
unit u_facto;

interface

procedure HelloWorld();

implementation

procedure HelloWorld();
begin
  WriteLn('Hello World!');
end;

end.
```

Maintenant que nous avons codé la fonction il faut l'appeler, c'est dans le `TPDelphi.dpr` que vous allez le faire. Celui-ci va donc contenir :

```
begin

  HelloWorld();
  ReadLn;

end.
```

Bravo, vous êtes maintenant prêts à coder des fonctions plus compliquées !

Chapitre 2

Factorielle

2.1 Définition de Factorielle

Pour une gestion plus simple des cas d'erreurs nous allons imposer une définition de factorielle spéciale qui vaut 1 pour des nombres négatifs.

$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n * (n-1)! & \text{si } n > 1 \end{cases}$$

Afin de pouvoir tester votre code, voici quelques exemples.

n	-42	-1	0	1	2	3	5	10
n !	1	1	1	1	2	6	120	3628800

★ **Conseil** Lorsque vous aurez à choisir des exemples par vous même par la suite, veillez à bien prendre en compte tous les cas particuliers (et à les implémenter) ! Ici il ne faut pas oublier les nombres négatifs, ni 0, ni 1.

2.2 Version Recursive

Nous allons débuter par une version récursive de la fonction qui à déjà été vue en Caml. Il s'agit d'écrire le code Delphi qui va retranscrire à la lettre la formule mathématique.

Prototype : `function fact_rec(n : integer) : integer;`

★ **Syntaxe** Vous devez placer la valeur de retour de la fonction dans la variable `Result`. Par exemple si votre fonction renvoie 0, vous aurez une ligne : `Result := 0;`

★ **Syntaxe** Pour réaliser des alternatives vous devez utiliser la structure `if then else`. Voici comment l'écrire

```
if (Condition) then
  Action1
else
  Action2;
```

Si la `Condition` est égale à `vrai` alors l'`Action1` est effectuée, sinon c'est l'`Action2` qui l'est.

2.3 Version Imperative - While

Delphi est un langage impératif à la base, ainsi pour en tirer parti nous allons transcrire cette fonction en impératif. Nous allons utiliser des boucles pour simuler les appels récursifs. Il existe plusieurs types de boucles mais la plus simple pour commencer reste la boucle `While`.

Prototype : `function fact_while(n : integer) : integer;`

★ **Syntaxe** La boucle while s'utilise de cette façon :

```
while (Condition) do
  Action
```

Tant que la `Condition` est vérifiée, alors on effectue l'`Action`. Avec la `Condition` étant une expression de type `Boolean`.

★ **Syntaxe** S'il faut effectuer plusieurs actions au sein de la boucle vous devez les encadrer de `begin` et `end`. Par exemple le code suivant va afficher les nombres de i a 0.

```
while (i >= 0) do begin
  WriteLn(i);
  i := i - 1;
end;
```

2.4 Version Imperative - For

Maintenant que vous avez réussi à retranscrire grâce à la boucle `While` la fonction factorielle, nous allons utiliser la boucle `For`. Lorsque l'on connait le nombre d'itérations à l'avance elle est plus pratique à utiliser.

Prototype : `function fact_for(n : integer) : integer;`

★ **Syntaxe** La boucle `For` s'utilise de cette façon :

```
for i := 1 to 10 do
  Action
```

Pour comprendre son fonctionnement, on peut la réécrire avec une boucle `While`. Vous remarquerez la concision de la boucle `for`.

```
i := 1;
while (i <= 10) do begin
  Action
  i := i + 1;
end
```

★ **Syntaxe** Nous voulons utiliser une boucle `For`, il nous faudra déclarer une variable `i` de type `integer`.

```
function fact_for(n : integer) : integer;
var
  i : integer;
begin
  for i := 1 to 10 do
```

★ **Attention** Vous devez initialiser vos variables si elles ne sont pas utilisées dans des boucles `For` (dans la majorité des cas) !

2.5 Version Imperative - Passage par valeur

Jusqu'à présent, nous avons utilisé des `function`s qui renvoient la valeur de la factorelle. Il est possible d'utiliser une `procedure` qui ne renvoie rien mais qui va modifier une des variables passées en paramètre.

Prototype : `procedure fact_val(n : integer; var fact : integer);`

Ainsi nous voulons à la place de placer le résultat dans la variable `Result` le placer dans la variable `fact`.

★ **Utilisation** Lorsque vous voudrez tester cette fonction, il va falloir déclarer une variable de type `integer` et la donner à la procédure. A la suite de l'appel, cette variable contiendra le résultat de l'opération.

```
var
  result_fact : integer;
begin
  fact_val(5, result_fact);
  // result_fact = 120
```

Chapitre 3

Point 2D

3.1 Type - Enregistrement

Nous allons apprendre à créer un nouveau type : un vecteur mathématique. Il comporte deux coordonnées représentées par deux champs dans notre enregistrement.

```
type
  pt2d = record
    x : integer;
    y : integer;
  end
```

★ **Syntaxe** Dans la majorité des cas, vous voulez placer le type dans la partie `interface` de votre unité. En effet, pour pouvoir exporter des fonctions qui utilisent ce type, il faut qu'il soit défini avant.

★ **Utilisation** Son utilisation est très simple. Il vous suffit de déclarer une variable du type `pt2d`. Ensuite pour accéder à chacun des champs il vous suffit de mettre un `.` devant son identifiant.

```
var
  a : pt2d;
begin
  a.x := 1;
  a.x := a.x + 1;
  writeln(a.x);
  // Affiche 2
```

3.2 Afficher un Point2D

La première chose à faire avant de coder quoi que ce soit est de s'assurer que l'on peut afficher ce que l'on est en train de faire. Nous allons donc réaliser une fonction qui prend un Point2D en paramètre, par exemple $\begin{pmatrix} x = 12 \\ y = -5 \end{pmatrix}$, et qui affiche dans la console :

```
x=12
y=-5
```

Prototype : `procedure print_pt2d(a : pt2d);`

★ **Attention** Lorsque l'on vous demande d'écrire quelque chose dans la sortie vous devez respecter le format à la lettre ! Pas d'espace autour du '=', pas les deux sur une même ligne ... Lorsque vous serez corrigés par une moulinette, la moindre faute de syntaxe vous vaudra 0.

3.3 Addition

Passons aux choses sérieuses. Vous devez coder l'addition de deux vecteurs, inutile de vous donner la formule ...

Prototype : `function pt2d_add(a, b : pt2d) : pt2d;`

3.4 Opposé

En prévision de la fonction de soustraction, nous allons coder une fonction qui donne l'opposé d'un Point2D, c'est-a-dire l'opposé de tous les champs. Par exemple

$$-\begin{pmatrix} x = 2 \\ y = -5 \end{pmatrix} = \begin{pmatrix} x = -2 \\ y = 5 \end{pmatrix}$$

Prototype : `function pt2d_opp(a : pt2d) : pt2d;`

3.5 Oppose par Valeur

Il faut savoir que lorsque l'on retourne un type structuré par une fonction, tous les champs sont recopiés dans une nouvelle variable. Même si dans cet exemple ce n'est pas important, lorsque ces opérations sont effectuées des millions de fois par seconde, toute optimisation, même aussi mineure que celle-ci, est bonne à prendre.

La méthode pour palier ce problème est le passage par valeur. En effet, la variable est directement modifiée et non copiée. Par contre il n'est plus possible d'utiliser l'ancienne variable.

Prototype : `procedure pt2d_opp_val(var a : pt2d);`

3.6 Soustraction avec Opposé

Maintenant que vous avez les deux fonctions Addition et Opposé, il va falloir les combiner afin de coder la fonction Soustraction.

Prototype : `function pt2d_sub(a, b : pt2d) : pt2d;`

3.7 Soustraction avec Opposé par Valeur

De même avec la fonction Opposé par Valeur.

Prototype : `function pt2d_sub_val(a, b : pt2d) : pt2d;`

3.8 Distance

Il est possible à partir de vecteurs d'obtenir d'autres types que des vecteurs. Par exemple on peut calculer la distance entre deux vecteurs qui est un réel.

$$dist(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Prototype : `function pt2d_dist(a, b : pt2d) : single;`

★ **Remarque** Il existe plusieurs représentations des nombres réels en Delphi. Les plus couramment utilises sont les `single` (généralement nommés `float`). Les `double` permettent de stocker de plus grandes valeurs. Quant aux `currency` qui permettent de stocker 4 décimales. Vous apprendrez le fonctionnement exact de chacun d'entre eux en cours.

★ **Remarque** Vous devrez utiliser la fonction racine carrée suivante :
`function sqrt(a : integer) : single;`

Chapitre 4

Bonus

Bravo, vous avez terminé le TP guidé, voici quelques exercices bonus pour parfaire vos notions de Delphi. Ils sont très similaires à ce que vous pourrez avoir pendant les partiels.

4.1 Histogramme

Le but de cet exercice est de vous faire manipuler les fichiers. Vous trouverez toute la documentation nécessaire sur le lien suivant :
`http://www.delphibasics.co.uk/Article.asp?Name=Files`

Vous devez écrire une fonction qui prend en entrée deux noms de fichiers. Elle lira le premier fichier et écrira dans le second un histogramme du nombre d'occurence de chaque chiffe.

Prototype : `procedure AfficherHistogramme(in, out : string);`

4.1.1 Histogramme Largeur

Voici un exemple pour l'histogramme en largeur.

```
Fichier Entree :

6 5 6 6 3 2 1 6 2 2 2 1 2 5 6 7 1 2 5 6 6 5 6 6 3 2 1 6 2

Fichier Sortie :

1 : * * * *
2 : * * * * * * * *
3 : * *
4 :
5 : * * * *
6 : * * * * * * * * * *
7 : *
```

4.1.2 Histogramme Hauteur

Plus dur, le même histogramme mais en hauteur.

```
Fichier Sortie :

1 2 3 4 5 6 7
- - - - - - -
* * *   * * *
* * *   * *
* *     * *
* *     * *
  *       *
  *       *
  *       *
  *       *
          *
          *
```

4.2 Recherche de chaîne

Le but de cet exercice est de vous faire manipuler les chaînes de caractères. Vous trouverez toute la documentation nécessaire sur le lien suivant : http ://www.delphibasics.co.uk/RTL.asp ?Name=String

Vous devez écrire une fonction qui prend en entrée trois noms de fichiers. Elle lira le premier fichier et écrira dans le dernier toutes les occurences des mots du premier dans le second.

Prototype : `procedure RechercheChaine(in1, in2, out : string);`

4.2.1 Recherche ligne par ligne

Dans une premiere partie, vous allez comparer les mots lignes par lignes.

```
Fichier Entree 1 :

fooo
vjeux
banban
epita
renard
```

```
Fichier Entree 2 :

epita
banban
belette
fooo
42
marmotte
delphi

Fichier Sortie :

fooo
banban
epita
```

4.2.2 Recherche complète

Il serait plus intéressant de donner des mots et de les rechercher dans un corpus de texte.

```
Fichier Entree 1 :

famille
belette
carotte
cactus
confondu
renard

Fichier Entree 2 :

La belette (Mustela nivalis) est le plus petit mammifère de la famille
des mustélidés et constitue également le plus petit mammifère
carnassier d'Europe avec une taille d'environ 20 cm pour une centaine
de grammes seulement. Vivant essentiellement dans les milieux
désertiques, la belette peut facilement être confondu avec une hermine.

Fichier Sortie :

famille
belette
confondu
```

Chapitre 5

Sponsor

Cette Conférence-TP a été réalisée et animée par la Fooo Team.