
Chapitre 2 LE LANGAGE PASCAL DE BASE



A la fin de cette partie, vous serez capable de:

- connaître la structure du langage Pascal (non objet);
- utiliser les divers types de données du Pascal;
- utiliser des instructions séquentielles, itératives et sélectives

2.1 La programmation structurée

Un ordinateur peut être assimilé à un système produisant des résultats à partir d'informations fournies et de "marches à suivre" permettant de les traiter. Les informations sont constituées par des données, et les méthodes de traitement par des algorithmes. Pour obtenir des résultats, la description des données et les algorithmes doivent être codés sous forme de programmes interprétables par l'ordinateur. En effet, le processeur de celui-ci ne peut exécuter qu'un nombre relativement restreint d'instructions élémentaires (le code machine).

Les programmes sont donc le résultat d'une succession d'étapes comprises entre la spécification informelle du problème et sa codification. Il y a ainsi entre ces deux pôles un "trou" qu'il s'agit de combler. Parmi les moyens ou les outils permettant d'y parvenir on peut citer notamment des environnements de production de logiciel (par exemple Delphi), des méthodes fournissant un encadrement au concepteur ou encore des langages de spécification permettant de préciser les étapes intermédiaires. Un autre aspect du rapprochement de la phase de codification vers la spécification du problème est constitué par le développement ou l'utilisation de langages de programmation permettant un niveau d'abstraction plus élevé.

Un des objectifs de la programmation structurée est la conception de logiciel fiable, efficace et d'une maintenance plus aisée. Il peut être atteint de manière asymptotique et par divers moyens. Les trois caractéristiques citées peuvent difficilement être évaluées dans l'absolu, car elles dépendent souvent de critères relatifs et subjectifs. Un programme n'est pas juste ou faux; sa qualité est une notion globale, constituée par plusieurs éléments, dont nous allons étudier les plus importants.

La fiabilité est une propriété informelle et parfois difficile à cerner. Cette propriété peut être atteinte grâce à deux qualités du langage de programmation. D'abord, la facilité d'écriture doit permettre d'exprimer un programme de façon naturelle ou en termes du problème à résoudre. Le programmeur ne doit pas être dérangé par des détails ou des habitudes du langage, mais doit pouvoir se concentrer sur la solution recherchée. Les langages modernes de haut niveau tendent vers cet objectif. Ensuite, la lisibilité du programme doit permettre d'en saisir aisément la construction logique et de détecter plus facilement la présence d'erreurs. Dans cette optique, l'instruction goto, par exemple, rend difficile la lecture du programme de façon descendante. Toutefois, dans certains cas, les objectifs énoncés au début de cette section peuvent être atteints dans de meilleures conditions par l'utilisation d'un goto (bien placé et bien documenté) plutôt que par une construction structurée; sa présence est alors acceptable. De telles situations sont toutefois extrêmement rares.

Si l'efficacité était au début l'objectif principal de la conception d'un programme, actuellement cette notion a évolué pour englober non seulement des critères de vitesse d'exécution et de place mémoire, mais aussi l'effort requis pour la maintenance du logiciel.

En effet, la nécessité de la maintenance impose au logiciel qu'il soit lisible et modifiable. Ces qualités sont souvent liées à des critères esthétiques. On peut néanmoins citer quelques facteurs qui facilitent les interventions dans un programme: un découpage approprié, une mise en page

claire, un choix adéquat des noms d'objets utilisés, des commentaires cohérents placés judicieusement.

2.2 Forme générale d'un programme Delphi

Dans Delphi on est peu confronté au programme proprement dit, mais plutôt à des unités, ce qui constitue une grande différence par rapport au Pascal traditionnel. De plus, le programme (principal), appelé projet:

- est généralement sauvegardé dans un fichier .dpr (pour Delphi Project);
- est toujours de petite taille;
- est automatiquement créé par Delphi;
- contient les références aux unités qui constituent l'application;
- initialise l'application, crée les différentes fiches et lance l'exécution de l'application.

En voici un exemple

```
program Project2;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};

{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Ainsi, la ligne marquée d'une flèche signifie que le programme utilise l'unité Unit1 stockée dans le fichier Unit1.pas et concerne la fiche Form1.

Lors du développement on est sans cesse en train de travailler dans des unités de Delphi. En voici un exemple:

```
unit Unit1;

interface

uses Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

Const MAX = 10;
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var Form1: TForm1;
    tab : array[1..MAX] of integer;

procedure Titre (taille: integer);
```

Partie
interface

Partie
implémentation

```
implementation  
  
uses Unit2;  
  
{ $R *.DFM }  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    form2.show;  
end;  
  
end.
```

Une unité comporte:

- un en-tête
- une partie interface dans laquelle figurent les objets exportables (visibles de l'extérieur de l'unité):
 - déclaration de constantes, de types (par exemple les objets) et de variables
 - des déclarations de procédures et/ou fonctions exportables
- une partie implémentation dans laquelle figurent les objets privés de l'unité, ainsi que le code des procédures et/ou fonctions déclarées dans la partie interface ainsi que de celles strictement privées;
- le corps de l'unité, ou partie d'initialisation, qui est souvent vide et réduite à "end."

En Pascal, il n'est pas possible de définir ou déclarer un objet à n'importe quel emplacement du programme. Ces définitions doivent être regroupées dans la partie réservée aux déclarations. De plus:

Tout objet référencé dans un programme doit avoir été préalablement défini.

Cette règle traduit une logique qui veut que des objets nouveaux soient construits uniquement à l'aide d'objets connus.

Enfin, on peut signaler une particularité intéressante: des déclarations de constantes, types et variables peuvent être placées entre les procédures. Dans ce cas, le domaine de visibilité de ces objets est constitué par la portion du programme située après ces déclarations. Cette possibilité permettant, par exemple, de disposer de variables "semi-globales" tend à rendre les programmes moins lisibles et va à l'encontre des principes de base liés à la notion de procédure.

2.3 Différents objets d'un programme

Si l'on répertorie les différents objets d'un programme, on obtient les catégories suivantes:

identificateurs

Chaque fois que l'on fait référence à un objet du programme (une variable, une constante, le nom d'une procédure...), c'est par l'intermédiaire d'un nom, appelé identificateur. Un identificateur est une suite de caractères de longueur non limitée dont le premier doit obligatoirement être une lettre (non accentuée). Après cette première lettre peuvent figurer exclusivement des chiffres, des lettres (non accentuées) ou des caractères de soulignement (dans un ordre quelconque). Le caractère de soulignement est souvent utilisé dans le but d'améliorer la lisibilité. Voici quelques exemples d'identificateurs:

```
A      Epsilon45   revenu_brut
```

Le choix des identificateurs peut également favoriser la lecture et la compréhension d'un programme. Il est déconseillé d'utiliser des identificateurs trop courts, ne suggérant aucune signification (par exemple A, X2, z), ainsi que des identificateurs trop longs, nécessitant plus de travail lors de l'écriture, et surtout n'offrant pas forcément une meilleure lisibilité.

Delphi ne distingue pas les minuscules des majuscules formant un identificateur. Ainsi les trois noms qui suivent désignent le même objet:

```
Rendement    rendement    RENDEMENT
```

mots réservés du langage

Il s'agit de mots ou de symboles qu'il n'est pas possible d'utiliser comme identificateurs déclarés dans le programme. En voici des exemples:

```
begin    program    and    until
```

constantes

Il s'agit, comme leur nom l'indique, d'objets qui gardent leur valeur tout au long de l'exécution d'un programme. Les constantes peuvent être de différents types, et constituées, entre autres, de nombres, de chaînes de caractères ou de caractères. Voici quelques exemples de constantes:

```
128      15.625      'A'      'Début'
```

identificateurs standard

Ce sont des identificateurs connus du langage Pascal, mais qui peuvent être redéfinis par l'utilisateur. On trouve parmi les identificateurs standard:

- les types standard

```
exemple:    integer    real    byte
```

- les procédures standard

```
exemple:    reset
```

- les fonctions standard

```
exemple:    sin    ord    chr    round    eoln
```

2.4 Constantes et variables

Le langage Pascal fait une distinction entre constantes et variables (ce qui n'est pas le cas pour des langages comme le BASIC). On utilise une constante chaque fois qu'un objet garde la même valeur tout au long d'un programme. Une constante reçoit une valeur au moment de la compilation du programme et cette valeur ne peut pas être modifiée. Une variable sera au contraire utilisée comme un objet dont la valeur peut être modifiée durant l'exécution du programme. Comme tous les objets définis par le programmeur, les constantes et les variables doivent être déclarées avant leur utilisation. Voici un exemple de déclaration de constantes et de variables:

```
procedure premier;
const nul = 0;
      code = 'secret';
      zede = 'z';
var age    : integer;
    salaire : real;
    sexe   : char;
...

```

Dans le cas des constantes, l'identificateur est suivi du signe "=" et de la valeur que l'on associe à la constante. Pour les variables, l'identificateur est suivi du signe ":" et du type de la variable.

Chaque déclaration de constante ou de variable se termine par un point virgule. Si plusieurs variables sont du même type, il est possible de les déclarer sous forme de liste d'identificateurs (séparés par une virgule) suivie du signe ":" et du type des variables:

```
var age, enfants, voitures : integer;
    salaire, taille : real;
```

Les déclarations précédentes peuvent également s'écrire:

```
var age      : integer;      { du père      }
    enfants  : integer;      { à charge   }
    voitures : integer;      { disponibles }
    salaire  : real;         { maximum    }
    taille   : real;         { du capitaine }
```

2.5 Commentaires

Dans le but d'améliorer la lisibilité et la compréhension des programmes, il est fortement conseillé d'introduire des commentaires. Un commentaire est un texte explicatif plus ou moins long, placé dans le programme, et ignoré par le compilateur. Les commentaires sont donc totalement invisibles et inutiles dans la phase de compilation et d'exécution d'un programme, mais d'une importance primordiale dans les phases de conception, de mise au point et de maintenance.

Ces explications, visibles uniquement dans les fichiers source (contenant le texte du programme), sont essentiellement destinées aux personnes susceptibles d'analyser un programme ou de lui apporter des modifications. Dans la plupart des cas, les commentaires sont destinés à l'auteur du programme. Mais dans tous les cas où un programme est réalisé en équipe, ou repris par d'autres personnes que l'auteur, les commentaires doivent être une aide à la compréhension, surtout dans certains passages peu évidents. Il n'est pas rare de rencontrer des programmes où les commentaires occupent plus de place que les instructions elles-mêmes. Il ne faut cependant pas oublier de mettre à jour les commentaires lors de la modification de tout ou partie du programme. Un commentaire devenu caduque ou qui n'a pas été mis à jour perd son utilité et peut même devenir un obstacle à la compréhension d'un programme.

En Pascal, les commentaires sont reconnus comme tels par le compilateur grâce à des marques de début et de fin qui sont soit des accolades { }, soit les symboles (* et *). Il est possible d'imbriquer un type de commentaire dans l'autre type de commentaire. En voici quelques exemples:

```
(* Programme écrit par :   Durant Eva           *)
{ imbrication de commentaires (* très drôles *) }
```

Delphi ajoute également un troisième type de commentaire issu du langage C: chaque ligne commençant par // est considérée comme un commentaire. Dans ce cas, le commentaire ne peut s'étendre que sur une ligne et ne possède qu'une marque de début:

```
//ceci est un commentaire du troisième type
```

2.6 Affectation

L'affectation (ou assignation) est l'une des instructions les plus importantes en Pascal. Elle permet de placer une valeur, qui est le résultat de l'évaluation d'une expression, dans une position mémoire référencée par une variable:

variable := expression;

où variable est l'identificateur d'une variable qui a été déclarée auparavant.

L'expression peut être une simple variable ou constante, ou bien une combinaison de constantes, variables et opérateurs arithmétiques. Le symbole := indique l'affectation et pourrait être traduit par "prend la valeur de". Les deux signes qui le composent doivent être accolés.

En Pascal, on a voulu éviter toute confusion entre le symbole de l'affectation et celui de l'égalité (représenté par le signe "="). Lors d'une affectation, l'expression qui se trouve à droite du symbole := est évaluée, ensuite seulement le résultat de l'évaluation est affecté à la variable située à gauche du symbole :=. De plus, il est impératif que le type de l'expression soit le même que le type de la variable à laquelle on affecte le résultat de l'expression.

2.7 Instructions et blocs d'instructions

Nous avons vu précédemment que les instructions sont séparées par des points virgules. Lorsque plusieurs instructions forment logiquement un tout, on est souvent amené à les grouper. On obtient alors un bloc d'instructions. Le début d'un tel bloc est indiqué par le mot réservé `begin`, alors que sa fin est indiquée par le mot réservé `end`. On parle parfois d'"instruction composée" au lieu de "bloc d'instructions". Ceci exprime bien le fait qu'un groupement de plusieurs instructions peut être vu comme une seule instruction (indivisible). Le corps d'un programme Pascal est lui-même un bloc d'instructions. Voici un exemple de bloc d'instructions:

```
begin
  age := 55;
  no := age * 10;
end;
```

Lorsque l'on écrit ses premiers programmes, les points virgules posent parfois des problèmes. En fait, ce qui peut sembler au début une contrainte, devient naturel après un temps d'adaptation. La règle concernant les points virgules est la suivante:

Chaque instruction doit se terminer par un point-virgule. Toutefois, le point-virgule peut être omis s'il est suivi des mots réservés `end` ou `until`. Il doit être omis s'il est suivi du mot réservé `else` (sauf s'il s'agit d'une structure sélective `case...of`).

En suivant cette règle, l'exemple précédent aurait pu s'écrire de la manière suivante

```
begin
  age := 55;
  no := age * 10
end;
```

Bien que cette forme d'écriture soit tout à fait correcte, il est conseillé de l'éviter au profit de la première citée. L'économie de points virgules peut parfois conduire à des erreurs de syntaxe lors de la modification d'un programme. Ajoutons, par exemple, une ligne à la fin du bloc d'instructions (avant le `end`). Le risque d'oublier de placer un point virgule à la fin de la ligne qui précède donne le résultat suivant:

```
begin
  age := 55;
  no := age * 10 (* il manque un ; ici *)
  no := no - age (* ligne ajoutée *)
end;
```

Ce fragment de programme n'est pas correct, car il manque le séparateur entre les deux dernières instructions. Le lecteur jugera lui-même, expérience faite, de l'opportunité de placer un séparateur d'instructions avant un `end` ou un `until`.

2.8 Types scalaires et expressions

Parmi les avantages qu'offre le langage Pascal, on trouve une grande richesse de types de données. Nous avons vu précédemment qu'à chaque variable correspond un type. La notion de type est très importante puisqu'elle détermine la nature et l'ensemble des valeurs que peut prendre une variable. Dans cette section nous nous intéressons aux types scalaires, caractérisés par un ensemble de valeurs ordonnées et non structurées.

Parmi les types scalaires on trouve les types entiers, réels, booléen et caractère, ainsi que les types énumérés ou définis par l'utilisateur. Nous étudierons également les expressions qu'il est possible de construire sur la base de chacun de ces types. Une expression est une combinaison d'objets qui sont des opérateurs, des opérandes et des parenthèses.

Nous serons amenés à évoquer la représentation interne des nombres, ou du moins la place mémoire occupée par une variable d'un type donné. En informatique, l'unité de mesure de la capacité mémoire est l'octet (byte); un octet étant lui-même composé de 8 chiffres binaires (bits) pouvant prendre chacun la valeur 0 ou 1.

Les types entiers

Il existe plusieurs type permettant de stocker des valeurs entière. Voici leurs caractéristiques:

Type	Intervalle	Format/taille
Byte	0..255	non signé, 1 octet
Word	0..65535	non signé, 2 octets
Shortint	-128..127	signé, 1 octet
Smallint	-32768..32767	signé, 2 octets
Integer	-2147483648.. 2147483647	signé, 4 octets
Cardinal	0.. 4294967295	non signé, 4 octets
Longint	-2147483648.. 2147483647	signé, 4 octets
Longword	0..4294967295	non signé, 4 octets
Int64	$2^{63}..2^{63}-1$	Signé, 8 octets

Tableau 2.1

Il convient de signaler que les types integer et cardinal sont stockés sur 2 octets sous Windows 3.x et sur 4 octets sous Windows 95 ou NT.

Malgré leurs particularités, les différents types entiers se comportent de la même manière. Pour garder une cohérence dans les types de données, une opération entre deux nombres entiers doit fournir un résultat de type entier. Ceci est effectivement le cas pour:

- l'addition $4 + 5$ donne 9
- la soustraction $12 - 20$ donne -8
- la multiplication $6 * 7$ donne 42

Mais la division pose un problème:

le quotient de 6 par 4 donne 1.5 qui n'est pas un nombre entier.

Il a donc fallu implémenter une division, spécifique aux nombres entiers, qui conserve le type des opérandes. En Pascal, cette division entière s'écrit **div** :

6 div 4 donne 1
10 div 3 donne 3

Le résultat de cette division correspond à la troncature de la partie décimale. Parallèlement à cette division entière, il est souvent utile de connaître le reste de la division entre deux nombres entiers. L'opérateur **mod** (abréviation de modulo) permet le calcul de ce résultat:

10 mod 3	donne 1
23 mod 4	donne 3

Lorsque des opérateurs et des opérandes sont combinés de manière à former une expression, il est nécessaire d'avoir une convention permettant de l'évaluer. Ainsi l'expression $4 * 2 + 3$ donnera-t-elle le résultat 11 ou bien 20 ? Tout dépend de l'ordre dans lequel sont effectuées les opérations. Les conventions d'évaluation en vigueur en Pascal correspondent à des règles de priorité, semblables à celles qui existent en mathématique:

- les opérateurs div, mod, et * sont prioritaires par rapport aux opérateurs + et - ;
- dans chacune de ces deux catégories les opérateurs ont la même priorité;
- en cas d'égalité de priorité, les opérations concernées sont effectuées de gauche à droite.

Ainsi:

$4 * 2 + 3$	donne 11
$8 + 4 * 3 \text{ div } 2$	donne 14
$6 \text{ mod } 4 * 2 \text{ div } 3$	donne 1

Pour modifier ces règles de priorité, il est toujours possible d'utiliser les parenthèses:

$4 * (2 + 3)$	donne 20
$7 \text{ div } ((5 \text{ mod } 3) \text{ mod } 4)$	donne 3

Il est également possible de faire précéder un nombre par l'opérateur unaire "-":

$-4 * 12$	donne -48
$4 * (-5)$	donne -20

Dans les programmes écrits en TURBO Pascal, les nombres entiers peuvent également être exprimés en notation hexadécimale, autrement dit en base 16. Cette possibilité est appréciée par certains programmeurs, car elle leur permet, dans des situations bien particulières, une représentation des nombres plus proche de celle rencontrée en langage machine.

Chaque fois qu'une constante numérique entière intervient dans une expression, il est possible de l'exprimer sous forme hexadécimale. Pour cela il faut placer le signe \$ immédiatement devant le nombre exprimé en base 16. Ainsi l'expression

```
total := 2 * 30;
```

peut s'écrire

```
total := $2 * $1E;
```

Cette notation est utilisable pour tous les types entiers.

Les types réels

Les différents types réels se différencient par leur domaine de définition, le nombre de chiffres significatifs (précision) et la place mémoire occupée. Le tableau suivant résume ces différentes caractéristiques:

Type	Intervalle	Chiffres	Taille en octets
Real48	$2.9 \cdot 10^{-39} \dots 1.7 \cdot 10^{38}$	11-12	6
Single	$1.5 \cdot 10^{-45} \dots 3.4 \cdot 10^{38}$	7-8	4
Double Real	$5.0 \cdot 10^{-324} \dots 1.7 \cdot 10^{308}$	15-16	8
Extended	$1.9 \cdot 10^{-4951} \dots 1.1 \cdot 10^{4932}$	19-20	10
Comp	$-2^{63} + 1 \dots 2^{63} - 1$	19-20	8
Currency	-922337203685477.5808.. 922337203685477.5807	19-20	8

Tableau 2.2

Conversion réels/entiers et entiers/réels

Il existe deux manières de convertir une valeur réelle en valeur entière: l'arrondi et la troncation. Le Pascal dispose de deux fonctions standard qui effectuent ces opérations sur les nombres réels.

La troncation

La fonction `trunc` agit de manière à tronquer la partie décimale d'un nombre réel. Son utilisation est illustrée par les exemples qui suivent:

```
trunc (4.2)           donne 4
trunc (4.99)         donne 4
trunc (-12.6)        donne -12
trunc (3.01)         donne 3
```

```
r := 5.67;           { r est de type réel }
i := trunc (r);     { i est de type entier et contient 5 }
```

L'arrondi

La fonction `round` permet d'arrondir un nombre réel à l'entier le plus proche:

```
round (2.99)         donne 3
round (2.5)          donne 3
round (2.499)        donne 2
round (-7.8)         donne -8
```

```
r := 4.77;           { r est de type réel }
i := round (r);     { i est de type entier et contient 5 }
```

Cette fonction possède également la propriété que `round(-x)` est équivalent à `-round(x)`.

Les fonctions prédéfinies s'expriment par un identificateur (le nom de la fonction) suivi d'un ou plusieurs arguments placés entre parenthèses. L'argument est la valeur transmise à la fonction en vue d'un traitement. Le résultat en retour est porté par l'identificateur de fonction et peut, par exemple, être affiché ou intervenir dans une expression.

```
Edit1.Text := IntToStr (trunc (3.6 * (round (1.4) - 2.5)));
```

La conversion d'une valeur entière en une valeur réelle s'effectue sans passer par des fonctions spécifiques. Lors de l'affectation d'une expression de type entier à une variable de type réel, la conversion de type est implicite. Dans les instructions qui suivent, a et b sont des variables de type entier et c de type réel:

```
a := 6;
b := 2;
c := 2 * a + b;          {c contiendra la valeur 14.0}
```

Très souvent le programmeur est confronté des expressions mixtes, contenant aussi bien des valeurs entières que réelles. Dans ce cas, le résultat sera de type réel.

```
c := 15 div a;          c contiendra 2.0
15 divisé par 6 donne 2 (division entière);
c := 15 / a;           c contiendra 2.5
la division "réelle" de deux entiers donne un résultat réel;
c := 1.5 * (a + b);    c contiendra 12.0
il s'agit d'une expression mixte dont le résultat est réel.
```

Type booléen (boolean)

L'ensemble des valeurs constituant le type booléen (**boolean**) est réduit à deux identificateurs de constantes prédéfinies: **true** (vrai) et **false** (faux). On parle souvent de variables ou d'expressions logiques, car le langage Pascal est fortement inspiré de la logique mathématique en ce qui concerne les opérations associées au type booléen. Les constantes et variables de ce type se déclarent de la manière suivante:

```
const demo = true;
var majuscules, termine : boolean;
```

La constante demo a la valeur true; les variables majuscules et termine peuvent recevoir les valeurs true ou false, selon leur utilisation dans le programme.

Parmi les opérateurs booléens on trouve, dans une première catégorie, les opérateurs relationnels. La liste qui suit présente les différents opérateurs relationnels utilisables en Pascal, avec leur symbole et leur signification:

<	plus petit
>	plus grand
=	égal
<>	différent
<=	inférieur ou égal
>=	supérieur ou égal

Ces opérateurs se rencontrent dans les expressions booléennes correspondant à des conditions. Comme nous le verrons plus loin, les conditions interviennent notamment dans les instructions sélectives et répétitives. Voici un exemple illustrant l'utilisation d'opérateurs relationnels:

```
begin
  ...
  vieux := age >= 60;
  ...
end.
```

Dans ce programme, `age >= 60` est une expression booléenne. Selon le contenu de la variable age, le résultat de cette expression est true ou false. La variable vieux étant de type booléen, elle peut recevoir ce résultat.

La seconde catégorie d'opérateurs booléens est constituée par des opérateurs purement logiques. Ils sont au nombre de quatre, et sont illustrés ci-dessous par des exemples:

not négation logique (non)
si $x > 12$ est vrai, alors not ($x > 12$) est faux

and conjonction logique (et)
($x > 2$) and ($x < 10$) est vrai, si ($x > 2$) est vrai et ($x < 10$) est vrai

or disjonction logique (ou)
($x < 2$) or ($x > 10$) est vrai, si ($x < 2$) est vrai ou ($x > 10$) est vrai

xor disjonction logique exclusive
a xor b est vrai si a et b n'ont pas la même valeur logique

Comme il est possible, et même très courant, de construire des expressions logiques contenant aussi bien des opérateurs arithmétiques que logiques, il convient d'étendre les conventions de priorité établies précédemment. Nous avons quatre classes d'opérateurs indiquées dans la liste qui suit, en partant de la plus prioritaire:

- 1) not
- 2) * / div mod and
- 3) + - or xor
- 4) < <= = <> >= >

Dans chacune des classes, les opérateurs ont la même priorité. En cas d'égalité de priorité, les opérations correspondantes sont effectuées de gauche à droite. Les parenthèses peuvent servir à forcer la priorité. Voici, à titre d'exemple, comment s'exprimerait en Pascal l'expression "somme est comprise entre 10 et 35":

```
(somme > 10) and (somme < 35)
```

Examinons deux autres exemples d'expressions booléennes:

```
age < date + 100
```

il s'agit ici de comparer le contenu de la variable age avec le résultat de date + 100. L'opérateur + a une plus grande priorité que l'opérateur <. De ce fait, cette expression booléenne est équivalente à age < (date + 100);

```
age < 40 and revenu > 6000
```

cette expression provoquera un message d'erreur de la part du compilateur, car elle présente un conflit de types. En effet, la priorité de l'opérateur and étant plus élevée que celle des opérateurs relationnels < et >, elle sera interprétée comme

```
age < (40 and revenu) > 6000
```

qui n'est pas une expression correcte.

En Pascal, l'affectation d'une expression booléenne à une variable de type booléen permet souvent d'économiser une instruction sélective if. Le programme qui suit est tout à fait correct:

```
var riche : boolean;  
    revenu : real;  
...  
    revenu := StrToFloat (Edit1.Text);  
    if revenu > 4000.0 then
```

```

        riche := true
    else
        riche := false;
    ...

```

mais il est plus concis d'écrire:

```

var riche   : boolean;
    revenu  : real;
    ...
    revenu := StrToFloat (Edit1.Text);
    riche := revenu > 4000.0;
    ...

```

Ces deux programmes sont équivalents et `riche` vaut `true` ou `false` selon la valeur introduite par l'utilisateur. Malheureusement la seconde forme est fréquemment négligée au profit de la première.

Type caractère (char)

Ce type dénote un ensemble de caractères, fini et ordonné. Chacun de ces caractères peut être exprimé grâce à son code ASCII. Une variable de type caractère (`char`) peut contenir un seul caractère, généralement spécifié entre apostrophes. Comme nous le verrons plus loin, il est possible de constituer des suites de caractères appelées chaînes de caractères. Voici un petit programme qui met en évidence l'emploi des variables et constantes de type caractère:

```

const effe   = 'F';
var  lettre : char;
     bip     : char;
    ...
     bip := chr(7);
     lettre := 'h';
     mot.text := effe;
    ...

```

Dans cet exemple, la constante `effe` contient une lettre majuscule indiquée entre apostrophes et la variable `bip` contient le caractère dont le code ASCII est 7. Ce code correspond à l'émission d'un bref signal sonore par le haut-parleur de l'ordinateur.

La fonction prédéfinie `chr` permet de référencer tous les caractères, y compris certains caractères du code ASCII qui ne sont pas affichables. L'argument de cette fonction est précisément le code ASCII du caractère désiré. Pour connaître le code des caractères disponibles, il convient de se reporter au manuel de référence de l'ordinateur utilisé.

Delphi dispose de deux facilités supplémentaires pour désigner les caractères. L'une est d'indiquer le symbole `#` suivi du code du caractère. Cette notation est équivalente à la fonction `chr` et permet, par exemple, d'incorporer des caractères particuliers dans une chaîne de caractères. L'autre concerne uniquement les caractères de contrôle (non affichables). On peut les spécifier en les faisant précéder du symbole `'^'`.

```
#27'G'
```

cette suite de caractères comprend le caractère correspondant à la touche `<Esc>`, suivi du caractère `G`. Cette séquence de caractères pourrait, si elle était par exemple envoyée à une imprimante, servir à placer celle-ci dans un mode d'impression donné;

```
if c = #13 then...
```

cette instruction permet de déterminer si la variable `c` (de type caractère) contient le caractère correspondant à la touche `<Return>`. On aurait également pu écrire:

```
if c = chr(13) then...
```

Type énuméré

En Pascal, l'utilisateur peut définir de nouveaux types de données, qui n'existent pas intrinsèquement dans le langage. On parle de types énumérés, ou types scalaires définis par l'utilisateur, ou encore types scalaires déclarés. Le programmeur spécifie lui-même l'ensemble des valeurs appartenant au type qu'il définit. Ces valeurs constituent une liste d'identificateurs de constantes que le programmeur doit indiquer dans la partie du programme réservée aux déclarations. Lors de la déclaration d'un type énuméré on indique, entre parenthèses, toutes les valeurs possibles constituant ce type. Voici comment définir un type énuméré dont les valeurs sont les quatre saisons:

```
type saisons = (printemps, ete, automne, hiver);
var periode : saisons;
```

Le type `saisons` est déclaré à l'aide du mot réservé **type**. Dès ce moment, il peut figurer dans des déclarations ultérieures, au même titre qu'un type prédéfini.

La possibilité de définir ses propres types de données et de les adjoindre aux types prédéfinis permet une plus grande souplesse du langage et du traitement des données. En voici un exemple:

```
...
type fruits = (pommes, poires, ananas, peches, amandes, noix);
   jours    = (lundi, mardi, mercredi, jeudi, vendredi,
               samedi, dimanche);
   langages = (VisualBasic, Delphi, Pascal, inconnu, Francais);
var dessert, fruit : fruits;
    aujourd'hui   : jours;
    langue        : langages;

...
dessert := amandes;
if aujourd'hui = dimanche then
    liste.items.add ('Congé');
...
```

fig. 2.3

Les types énumérés sont toutefois soumis à un certain nombre de restrictions. Par exemple, la lecture et l'affichage des valeurs figurant dans un type énuméré ne sont pas autorisés, car il ne s'agit pas de chaînes de caractères, mais d'identificateurs. Une autre contrainte est qu'une même valeur ne peut pas figurer dans deux listes différentes, c'est-à-dire dans deux types énumérés différents. Ainsi, la déclaration suivante n'est pas autorisée:

```
type appareils = (television, aspirateur, transistor,
                 ventilateur);
   electronique = (resistance, condensateur, diode,
                  transistor);
```

En revanche, les possibilités offertes sont intéressantes. Comme pour tous les types scalaires (sauf le type réel), les valeurs des types énumérés sont ordonnées et dénombrables, ce qui rend possible l'utilisation des fonctions prédéfinies **pred** (prédécesseur), **succ** (successeur) et **ord** (numéro d'ordre). Les valeurs d'un type énuméré sont ordonnées en fonction de l'ordre dans lequel elles apparaissent lors de la déclaration, la première valeur d'un type porte le numéro d'ordre 0. Voici quelques exemples, se référant aux déclarations de la figure 2.3, qui illustrent l'emploi de ces nouvelles fonctions en relation avec les types entiers, booléen, caractère et énuméré:

```
ord (6)           vaut 6
ord ('A')         vaut 65
succ ('b')        vaut 'c'
pred (167)        vaut 166
```

ord (mardi)	vaut 1
pred (dimanche)	vaut samedi
succ (lundi)	vaut mardi

mais aussi:

ord (ord (mercredi))	vaut 2
pred (pommes)	n'est pas correct
succ (Francais)	n'est pas correct
pred (succ(poires))	vaut poires
succ (pred(poires))	vaut poires
pred (pred(dimanche))	vaut vendredi
pred (true)	vaut false
succ (false)	vaut true
ord (false)	vaut 0

Lorsqu'un type énuméré compte 256 valeurs ou moins, ces dernières sont représentées par un nombre de type byte. Dans le cas contraire, elles le sont par un nombre de type word. La fonction pred appliquée au premier élément d'un type ordinal ou la fonction succ appliquée au dernier élément d'un type ordinal ne fournissent pas un résultat correct.

Du fait qu'elles sont ordonnées, les valeurs d'un type énuméré peuvent être comparées à l'aide des opérateurs relationnels. Les instructions sélectives et répétitives suivantes tirent profit de cette possibilité:

```

if dessert >= amandes then
  liste.items.add ('Fruits secs');

if aujourd'hui < samedi then
  liste.items.add ('Jour ouvrable');

while (langue = Pascal) or (langue = Francais) do ...

for fruit:=pommes to peches do
  if dessert = fruit then
    liste.items.add ('Ce n''est pas un fruit sec');

repeat
  fruit := succ (fruit);
until fruit = noix;

```

Type intervalle

Nous avons vu qu'à chaque type de données est associé un ensemble de valeurs. Les variables déclarées avec un type donné peuvent prendre uniquement des valeurs correspondant à ce type. Mais il est fréquent d'utiliser seulement une portion de l'ensemble des valeurs possibles. Dans ce cas, on pourrait restreindre cet ensemble de valeurs à l'intervalle qui nous intéresse. Le langage Pascal permet cette restriction en faisant appel au "type" intervalle, qui n'est pas un type au sens strict, mais un sous-ensemble de valeurs prises dans un type de base. Ce type de base peut être un type scalaire quelconque, à l'exception d'un type réel. L'exemple qui suit illustre de quelle manière l'étendue d'un type peut être restreinte à un intervalle:

```

type lettre          = 'A'..'Z';
   entre_deux_guerres = 1919..1938;
   minuscules        = 'a'..'z';

var caractere : lettre;
   date       : entre_deux_guerres;
   min        : minuscules;

```

fig. 2.4

Le type `lettre` est un intervalle défini par rapport au type `caractere`. Donc, seuls les caractères compris entre 'A' et 'Z' peuvent être affectés à la variable `caractere`. De même, la variable `date` pourra prendre uniquement des valeurs entières comprises entre 1919 et 1938. Les bornes de l'intervalle sont incluses dans l'ensemble des valeurs possibles.

Le type intervalle est essentiellement utilisé dans deux buts. D'abord pour améliorer la lisibilité et la compréhension, mais également afin d'augmenter la fiabilité des programmes, car le Pascal détecte si une variable reçoit une valeur hors de l'intervalle déclaré. Dans l'exemple de la figure 2.4, il est sous-entendu que la variable `min` ne contiendra que des lettres minuscules. Si, au cours de l'exécution du programme, cette variable reçoit une autre valeur, une erreur sera signalée. Ce genre de problème devrait inciter le programmeur à revoir son programme et éventuellement à corriger une erreur de logique.

Le type énuméré sert souvent de type de base au type intervalle. L'exemple qui suit en est une illustration:

```

type jours          = (lundi, mardi, mercredi, jeudi, vendredi, samedi,
                       dimanche);
   week_end        = samedi..dimanche;

var aujourd'hui    : jours;
   conge           : week_end;
   travail         : lundi..vendredi;

```

L'utilisation du type intervalle entre dans le cadre de la discipline que le programmeur s'impose pour faciliter une intervention ultérieure sur son programme. Cet effort supplémentaire est souvent récompensé par la réalisation de programmes plus sûrs et plus lisibles.

Conversion de types scalaires

Les valeurs de type scalaire peuvent être converties en valeurs entières à l'aide de la fonction `ord`. Le Pascal offre également la possibilité d'effectuer la conversion d'une valeur de type scalaire en une valeur d'un autre type scalaire. Par exemple, considérons les déclarations suivantes:

```

type couleurs = (jaune, bleu, rouge, vert);
   formes     = (carre, triangle, cercle);
var teinte    : couleurs;
   figure     : formes;
   entier     : integer;

```

Il est possible d'effectuer des conversions de types en spécifiant l'identificateur du type désiré suivi d'un paramètre entre parenthèses. Ce paramètre doit être une valeur appartenant à un type scalaire connu ou déclaré:

```

entier := integer (rouge);      { = 2          }
figure := formes (bleu);       { = triangle  }
entier := integer ('7');       { = 55        }
teinte := couleurs (0);        { = jaune     }
figure := formes (0);          { = carre     }

```

Bien que cette conversion de types soit rarement utilisée, il convient de la signaler.

D'une manière générale, en Pascal, il n'est pas permis de mélanger différents types de données. Ce principe présente des exceptions lorsque deux types sont compatibles. C'est le cas, par exemple, pour les différents types entiers, qui sont compatibles à condition que les valeurs soient cohérentes par rapport à leur ensemble de définition. Les principes suivants régissent la compatibilité des types:

- les règles de cohérence doivent être respectées lors du mélange de données de type entier ou réel. Une expression mixte, comprenant des opérandes entiers et réels fournit un résultat de type réel;
- le mélange de données de type numérique (entier ou réel) et de type caractère n'est pas autorisé.

Structures séquentielles

Une structure séquentielle est une suite d'instructions qui s'exécutent les unes à la suite des autres, en séquence:

```
begin
  temperature := 28;
  meteo.caption := 'Il fait chaud';
end;
```

Structures sélectives

Ces structures permettent d'effectuer des choix selon des critères (ou conditions) que le programmeur a fixés. Ces instructions se comportent comme un aiguillage, à deux ou plusieurs branches. Selon qu'un critère est satisfait ou non, l'exécution du programme se poursuivra dans une "branche" ou dans une autre.

Instruction if

Considérons l'exemple suivant:

```
var temperature : integer;
...
temperature := StrToInt (Edit1.text);
if temperature > 20 then
  Label1.Caption := 'Il fait chaud'
else
  Label1.Caption := 'Il fait froid';
...

```

fig. 2.5

Selon la valeur indiquée par l'utilisateur et placée dans la variable `temperature`, le programme affichera "Il fait chaud" ou bien "Il fait froid". On remarque également les trois mots réservés **if**, **then** et **else** utilisés dans cette structure sélective qui, traduite en français, s'exprimerait par:

« si la température est supérieure à 20 alors afficher qu'il fait chaud
 sinon afficher qu'il fait froid »

L'exemple qui suit utilise une structure sélective sous une forme quelque peu différente:

```
...
var revenu : real;
  taxes : real;
...

```

```

taxes := revenu * 0.053;
if revenu > 4000.0 then
    revenu := revenu - taxes;
...

```

fig. 2.6

Traduite en français, la structure sélective signifie:

« si le revenu est supérieur à 4000.0 alors on déduit 5,3 % du revenu »

Dans l'exemple de la figure 2.5, le choix s'effectue entre deux instructions, l'une des deux étant forcément exécutée. Alors que dans le second exemple (figure 2.6) il s'agit d'exécuter ou de ne pas exécuter une instruction, à savoir la déduction des taxes.

Une "instruction" peut être constituée par une seule instruction ou par un bloc d'instructions. Illustrons cela par l'exemple suivant:

```

...
var revenu  : real;
    taxes   : real;
...
if revenu > 4000.0 then
begin
    revenu := revenu - (revenu * 0.053);
    msg := 'Le revenu subit des déductions.';
end else
    msg := 'Le revenu ne subit pas de déductions.';
...

```

Dans ce programme, deux instructions doivent être exécutées si le revenu est supérieur à 4000 francs. Il faut donc grouper ces deux instructions en un bloc délimité par **begin** et **end**. Ce concept est général en Pascal:

A chaque endroit d'un programme où une instruction peut figurer, il est possible de la remplacer par un bloc d'instructions.

Comme une structure sélective est elle-même une instruction, on peut emboîter plusieurs structures `if...then...else`:

```

if x > 0 then positif := positif + 1
    else if x < 0 then negatif := negatif + 1
        else zero := zero + 1;

```

L'une des branches d'une instruction sélective peut donc contenir une autre instruction sélective. Ces structures emboîtées contribuent, dans le cas de l'instruction `if...then`, à rendre un programme moins lisible. L'instruction qui suit semble être ambiguë; à quel `if` se rapporte le `else`?

```

if x > 0 then if y = 3 then z := y else z := x;

```

En fait, les langages de programmation n'admettent pas les ambiguïtés. Dans notre exemple, l'ambiguïté est levée par une convention stipulant qu'un `else` se rapporte toujours au `if` précédent qui est le plus proche. Il convient, dans tous les cas, de soigner la présentation d'un programme; une indentation (décalage par rapport à la marge gauche) convenable augmente la lisibilité du programme. L'exemple précédent peut également s'écrire:

```

if x > 0 then if y = 3 then z := y
                else z := x;

```

ou encore:

```
if x > 0 then
  if y = 3 then
    z := y
  else
    z := x;
```

Le Pascal permet de "court-circuiter" l'évaluation d'expressions booléennes contenant des opérateurs `and` et/ou `or`. Pour mieux comprendre l'importance de la manière dont une telle évaluation est effectuée, considérons l'instruction conditionnelle suivante:

```
if (b <> 0) and (a / b > 4) then ...
```

S'agissant d'une expression booléenne faisant intervenir un "et" logique, il suffit que l'un des deux membres de l'expression prenne la valeur "faux" pour que l'expression entière soit "fausse". Dans le cas où `b` possède la valeur 0, le Pascal n'évalue pas le second membre de l'expression, évitant ainsi une erreur à l'exécution due à une division par zéro. Cette particularité peut s'avérer fort utile dans certaines situations, et, dans tous les cas, elle augmente la vitesse d'exécution des programmes.

Il est néanmoins possible d'éviter ce type de raccourci dans l'évaluation des expressions booléennes grâce à une directive fournie au compilateur.

Instruction case

Considérons maintenant la partie d'un programme qui fait bouger un point sur l'écran. Supposons que l'utilisateur appuie sur les lettres 'H' pour "haut", 'B' pour "bas", 'D' pour "droite", 'G' pour "gauche". Cette partie de programme pourrait s'écrire de la manière suivante:

```
...
if key='H' then y := y - 1;
if key='B' then y := y + 1;
if key='D' then x := x + 1;
if key='G' then x := x - 1;
im.canvas.pixels[x, y] := clRed;
...
```

ou bien encore:

```
...
if key = 'H' then y := y - 1
else if key = 'B' then y := y + 1
  else if key = 'D' then x := x + 1
    else if key = 'G' then x := x - 1;
im.canvas.pixels[x, y] := clRed;
...
```

Dans ces exemples, les structures sélectives se ressemblent et paraissent un peu lourdes dans leur notation. Le langage Pascal dispose d'une autre structure sélective permettant d'éviter dans certains cas de telles situations. Cette nouvelle structure correspond à l'instruction **case ... of**, et son utilisation permet de modifier l'exemple précédent en:

```
...
case key of
  'H' : y := y - 1;
```

```

    'B' : y := y + 1;
    'D' : x := x + 1;
    'G' : x := x - 1;
end;
im.canvas.pixels[x, y] := clRed;
...

```

Cette nouvelle structure agit en fait à la manière d'un test à multiples branches. Pour chaque branche l'égalité entre la valeur de ch (appel, sélecteur) et la constante correspondante est vérifiée. Si effectivement la valeur de ch équivaut à une des constantes, l'instruction ou le bloc d'instructions correspondant sont exécutés. Si la valeur du sélecteur ne correspond à aucune des constantes indiquées l'exécution du programme se poursuit après la structure sélective.

Cette structure est commune à toutes les implémentations de Pascal. Delphi offre cependant une variante parfois utile: une branche supplémentaire indiquée par un **else** permet l'exécution d'une instruction ou d'un bloc d'instructions au cas où la valeur du sélecteur ne correspond à aucune des constantes spécifiées. L'exemple qui suit illustre cette possibilité:

```

...
case key of
  'H' : y := y - 1;
  'B' : y := y + 1;
  'D' : x := x + 1;
  'G' : x := x - 1;
else
  msg := 'Erreur';
end;
im.canvas.pixels[x, y] := clRed;
...

```

Remarquons qu'avant le mot réservé **else** d'une structure **case** on trouve un point virgule

Dans chaque branche d'une instruction **case** on peut indiquer plus d'une valeur, et même un intervalle de valeurs. L'exemple qui suit illustre cette possibilité:

```

case a + b of
  1           : a := b;
  3..6       : b := a;
  8, 9       : begin
                a := 0;
                b := 0;
              end;
  10..13, 15 : b := 0;
end;

```

Structures itératives

Introduction

Arrivés à ce point, nous ne sommes pas encore en mesure d'écrire un programme qui répète certaines instructions. Il nous manque les structures itératives qui permettent d'effectuer ce que l'on appelle communément des boucles. Dans le langage Pascal, on trouve trois types d'instructions répétitives. Les deux premières se distinguent par le fait que la condition de sortie est conséquente à l'évaluation d'une expression booléenne. La troisième est liée à l'évaluation implicite d'un compteur de boucle.

Instruction while

Cette structure permet de répéter l'exécution d'une instruction ou d'un bloc d'instructions tant qu'une expression est vérifiée:

```
...
while nombre < 10000 do
begin
    Edit1.text := Edit1.text + '1';
    nombre := nombre * 10;
end;
...
```

fig. 2.7

Dans cet exemple, tant que le nombre est inférieur à 10000, le chiffre 1 est concaténé.

Si le nombre vaut, par exemple, 20000 avant l'entrée dans la boucle, le programme n'effectue pas les instructions contenues dans la structure **while**. Il est donc possible que les instructions englobées par ce type de boucle ne soient jamais exécutées.

Instruction repeat

Cette instruction permet de répéter l'exécution d'une instruction ou d'un bloc d'instructions jusqu'à ce qu'une condition soit vérifiée. En apparence, cette structure ressemble à la précédente. En réalité, la différence est significative et d'importance. Reprenons l'exemple de la figure 2.7 et voyons comment on pourrait l'écrire à l'aide d'une instruction repeat:

```
...
repeat
    Edit1.text := Edit1.text + '1';
    nombre := nombre * 10;
until nombre >= 10000;
...
```

fig. 2.8

Le fonctionnement des programmes illustrés sur les figures 2.7 et 2.8 est pratiquement identique. La différence fondamentale concerne l'emplacement de la condition par rapport au contenu de la boucle. Dans le second cas, même si le nombre est déjà supérieur à 10000, les instructions contenues dans la structure **repeat** seront exécutées une fois. La condition qui détermine l'éventuel arrêt de la répétition (nombre >= 10000) se trouvant à la fin de la boucle, l'entrée dans la structure est obligatoire. De plus, dans l'exemple de la figure 2.8, l'expression booléenne constituant la condition d'arrêt est la négation logique de l'expression booléenne de la figure 2.7. En effet, dans le cas d'une boucle **while**, il s'agit d'une condition de continuation.

Instruction for

Cette troisième structure répétitive est utilisée lorsque le nombre d'itérations est connu. Il faut spécifier l'identificateur d'une variable appelée indice dont la valeur est modifiée implicitement au cours de l'exécution de la boucle, la première valeur prise par cette variable ainsi que la valeur pour laquelle la répétition s'arrête. A chaque passage dans la boucle, l'indice prend la valeur suivante ou précédente (selon si la boucle est ascendante ou descendante). Le programme qui suit affiche les carrés des nombres entiers compris entre 1 et 10:

```
...
const max = 10;
var    nombre : integer;
begin
    for nombre := 1 to max do
        liste.items.add (IntToStr (nombre * nombre));
    end;
```

Les résultats fournis par ce programme se présentent sous la forme d'un ListBox dont le contenu est:

```
1
4
9
16
25
36
49
64
81
100
```

Le mot réservé **to** peut être remplacé par **downto** afin que la boucle soit parcourue dans l'ordre décroissant de l'indice (dans l'exemple, la variable nombre). La boucle s'écrit alors:

```
for nombre := max downto 1 do ....
```

et les résultats seront:

```
100
81
..
```

Il n'est pas possible, en Pascal, d'indiquer explicitement un pas d'incréméntation de la boucle. En indiquant **to**, l'indice prend la valeur suivante, alors qu'en utilisant **downto** il prend la valeur précédente. Pour un pas différent, il convient d'utiliser une boucle **while** ou **repeat**.

Le Pascal ne permet pas non plus d'utiliser la boucle **for** avec un indice de type réel. Il faudra dans ce cas aussi faire appel à une boucle **while** ou **repeat**. En revanche, l'indice d'une boucle **for** peut être de type entier, énuméré, booléen ou caractère.

Lorsqu'une instruction **for** se trouve dans une procédure ou une fonction, la variable constituant l'indice de la boucle doit obligatoirement être déclarée localement à la procédure ou à la fonction.

Procédure Halt

La procédure **Halt** provoque une fin anormale d'un programme et passe le contrôle au système d'exploitation. Pour provoquer la fin normale d'une application il convient d'utiliser **Application.Terminate**.

On peut facultativement passer un nombre entier comme paramètre à la procédure **Halt**, comme dans l'exemple qui suit. Il s'agit d'un code de sortie transmis au système d'exploitation. Ce code est laissé au choix du programmeur.

```
begin
  if 1 = 1 then
    if 2 = 2 then
      if 3 = 3 then
        Halt(1); { On quitte le programme }
        Form1.caption := 'Ce code ne sera jamais exécuté';
      end;
    end;
  end;
```

Procédure Break

La procédure **Break** provoque l'interruption d'une boucle **for**, **while** ou **repeat**.

L'exemple qui suit montre une procédure qui teste si un nombre **nb** est premier. Elle comporte une boucle **for** dans laquelle on détermine si le nombre **nb** est divisible par les nombres compris entre **2** et **nb - 1**. Dès que l'un de ces nombres divise **nb** cela signifie que **nb** n'est pas premier. Il est donc inutile de poursuivre l'exécution de la boucle, d'où l'utilisation de la procédure **Break**.

```
procedure TForm1.testClick(Sender: TObject);
var nb      : integer; // nombre à tester
    i      : integer; // indice de boucle
    premier : boolean;
begin
    premier := true;
    nb := strtoint (nombre.text);
    for i := 2 to nb - 1 do begin
        if nb mod i = 0 then begin
            premier := false;
            break;
        end;
    end;
    if premier then
        resultat.text := 'Nombre premier'
    else
        resultat.text := 'Nombre non premier';
end;
```

Procédure Continue

L'appel à la procédure **Continue** provoque le passage du contrôle de l'exécution à l'itération suivante dans une instruction **for**, **while** ou **repeat**.

```
procedure traitement;
var i : integer;
begin
    for i := 0 to MAX do begin
        if t[i] = 0 then
            continue; { on passe à l'itération suivante }
        ...
        { ici les instructions de la boucle si t[i] <> 0 }
        ...
    end;
end;
```

Procédure Exit

La procédure **Exit** permet de quitter l'exécution de la procédure en cours. Si la procédure en cours correspond au programme principal, **Exit** termine l'exécution du programme. L'utilisation de la procédure **Exit** permet souvent de rendre le programme plus clair en évitant la lourdeur des clauses **else**. L'utilisation de la procédure **Exit** dans le fragment de programme suivant permet, entre autre de ne pas utiliser de **else** et ne nuit aucunement à la lisibilité du code, au contraire:

```

procedure TForm1.AmisClick(Sender: TObject);
var nb : integer;
    divnb1 : integer;
    divnb2 : integer;
    min, max : integer;
begin
  if strtoint (inf.text) >= strtoint (sup.text) then begin
    showmessage ('Limite supérieure > limite inférieure SVP');
    exit;
  end;
  listbox1.clear;
  min := strtoint (inf.text);
  max := strtoint (sup.text);
  for nb := min to max do begin
    divnb1 := SommeDiviseurs (nb);
    divnb2 := SommeDiviseurs (divnb1);
    if (divnb2 = nb) then
      listbox1.items.add (inttostr(nb) + ' et ' + inttostr(divnb1)
        + ' sont amis');
  end;
end;

```

Instruction goto

Cette instruction correspond à un branchement inconditionnel et permet de rompre une séquence en poursuivant l'exécution d'un programme à une autre instruction que la suivante. L'emplacement de l'instruction de branchement est repéré par une étiquette, qui n'est autre qu'un identificateur ou un numéro compris entre 0 et 9999. Une étiquette doit être déclarée après le mot réservé **label** dans la partie réservée aux déclarations. Son domaine de définition correspond au bloc où elle est déclarée; il n'est donc pas possible d'effectuer des branchements par l'instruction **goto** entre procédures ou fonctions. Si une déclaration d'étiquette figure dans un programme, cette étiquette doit impérativement être utilisée.

L'exemple qui suit montre comment s'utilise cette instruction:

```

procedure demo;
label ici;
...
begin
  if Label1.left > 150 then
    goto ici
  else
    Label1.caption := 'Fin';
  ...
  ici:
  ...
end;

```

Ce programme est bien entendu construit de manière aberrante. Il suffit de l'écrire comme suit pour que l'instruction goto n'ait plus de raison d'être:

```

procedure demo;
...
begin
  if Label1.left <= 150 then begin
    Label1.caption := 'Fin';
    ...
  end;
  ...
end;

```

L'emploi de cette instruction en Pascal peut presque toujours être évité, bien que dans de très rares cas elle soit fort utile (par exemple en tant que sortie rapide d'une boucle complexe). Si l'on se réfère au théorème de structures, l'utilisation de l'instruction **goto** n'est jamais nécessaire.

EXERCICES

Exercice 2.1:

Exercice 2.1 Calculs

Premier nombre: 360

Deuxième nombre: 5

Somme 365

Différence 355

Produit 1800

Quotient 72 Reste 0

Quotient 72.0000

Calculs Fin

Reproduire la fiche ci-dessus. L'utilisateur peut entrer deux nombres **entiers**, faire effectuer les calculs et quitter le programme.

Améliorations:

- Refuser les divisions par 0
- Enlever les résultats lorsqu'ils ne sont plus justes
- Refuser les caractères illicites pour les 2 nombres

Exercice 2.2:

	Quantité	Prix	Total
Chaise modèle 456.7	<input type="text"/>	<input type="text" value="67.50"/>	<input type="text"/>
Table ancienne	<input type="text"/>	<input type="text" value="985.50"/>	<input type="text"/>
Chariot à roulettes	<input type="text"/>	<input type="text" value="275.00"/>	<input type="text"/>
Total			<input type="text"/>
Rabais	<input type="text"/>	%	<input type="text"/>
Total net			<input type="text"/>

Reproduire la fiche ci-dessus.

L'utilisateur ne peut saisir que les quantités, les autres Edits sont en ReadOnly ou désactivés.

Le rabais dépend du montant total:

Si le total est inférieur à 1'000 Fr, le rabais est de 2 %

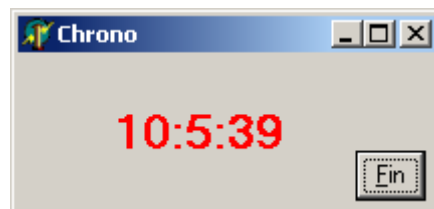
Si le total est compris entre 1'000 Fr et 5'000 Fr, le rabais est de 3.5 %

Si le total est compris entre 5'000 Fr et 10'000 Fr, le rabais est de 5 %

Si le total est supérieur à 10'000 Fr, le rabais est de 7 %

Exercice 2.3:

Ce programme doit simuler un chronomètre.



Le temps au départ est fixé par le programmeur. Tant que l'utilisateur n'a pas appuyé sur le bouton Fin, le chronomètre avance.

Il faut utiliser 3 variables globales (heures, minutes, secondes).

La difficulté est de passer de 59 secondes à une minute supplémentaire et de 59 minutes à une heure supplémentaire.



Pour mesurer le temps qui passe, il faut utiliser un Timer (onglet Système)

Propriétés:

- **Enabled** utilisée pour activer (true) ou désactiver (false) le Timer
- **Interval** détermine l'intervalle de temps, exprimé en millisecondes, s'écoulant avant que le composant timer génère un autre événement OnTimer.
- **Name** nom du composant

Événement:

- **OnTimer** événement utilisé pour écrire un gestionnaire d'événement qui exécute des actions à intervalles réguliers. La propriété Interval détermine la périodicité des événements OnTimer. A chaque fois que l'intervalle spécifié s'écoule, l'événement OnTimer a lieu.

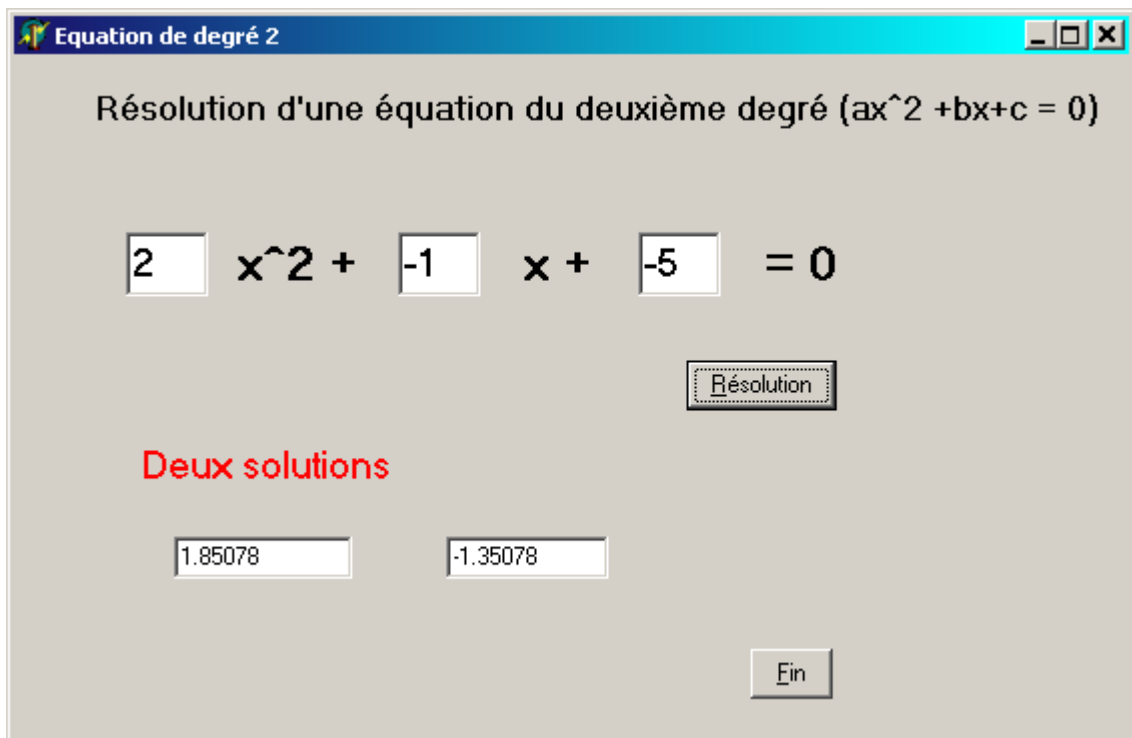
Autre possibilité: refaire le programme en utilisant uniquement une variable représentant les secondes. Les minutes et les heures seront obtenues en faisant appel à **div** et **mod**.

Exercice 2.4:

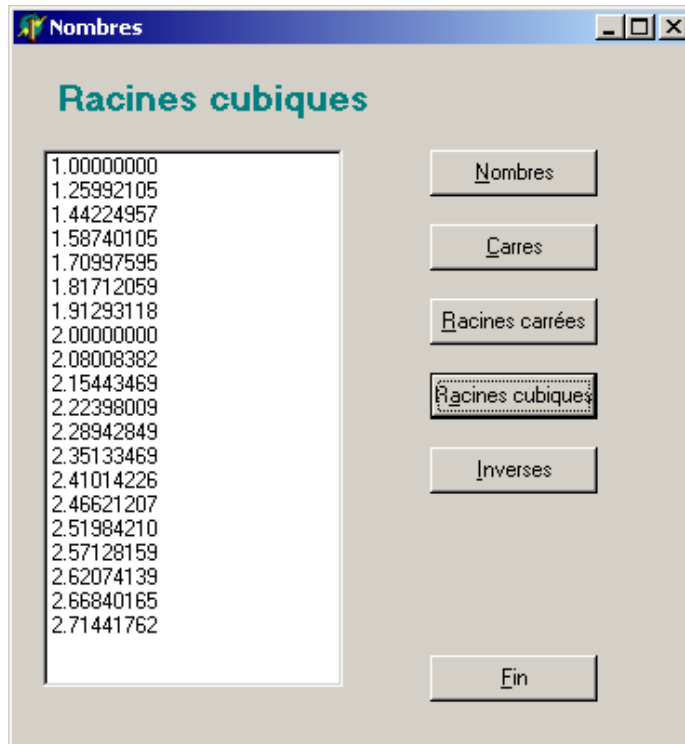
Ecrire un programme qui résout une équation de degré 2.

$$ax^2 + bx + c = 0$$

L'algorithme est discuté en classe.



Exercice 2.5:



Ce programme affiche, dans un ListBox, les nombres de 1 à 20, ou bien leur carré, leur racine carrée, leur racine cubique, leur inverse.

Les seules propriétés à connaître concernant les ListBox sont:

- **Clear** efface le contenu du ListBox
- **Items.Add** ajoute une ligne après la dernière ligne du ListBox, l'argument doit être une chaîne de caractères

Exemple:

```
Liste.Clear;  
Liste.Items.Add ('Ceci est la dernière ligne');  
Liste.Items.Add (IntToStr (nombre));
```

Utilisez l'aide de Delphi pour trouver comment extraire une racine cubique.

Amélioration:

Afficher chaque fois le nombre en regard de sa racine, de son inverse ou de son carré.

Exercice 2.6:

Intérêts composés et capital final

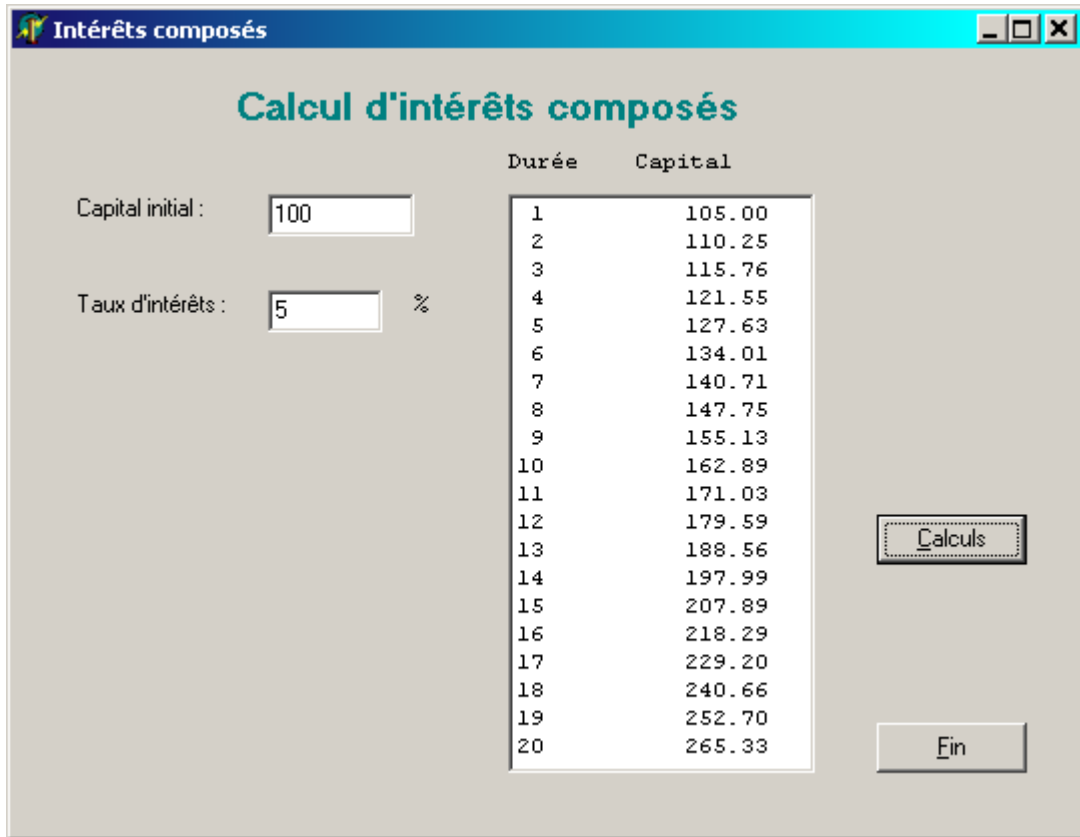
$$C = C_0 * (1 + T)^N$$

Où C_0 est le capital initial en francs

T est le taux d'intérêts en valeur décimale

N est le nombre d'années de placement

L'utilisateur doit indiquer le capital initial et le taux d'intérêts en % et le programme affiche ce que devient ce capital, chaque année, s'il placé durant 20 ans.



Exercice 2.7:

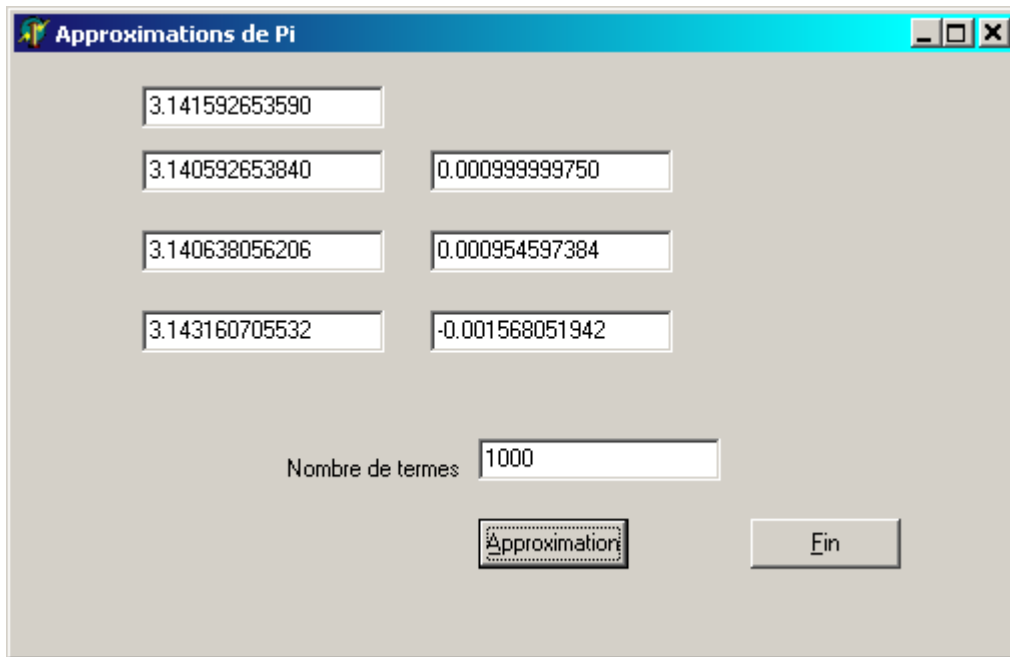
Ecrire un programme qui approxime pi de 3 manières différentes:

$$\frac{1}{3.5} + \frac{1}{7.9} + \dots + \frac{1}{(4k-1)(4k+1)} \rightarrow \frac{1}{2} - \frac{\pi}{8}$$

$$1 + \frac{1}{4} + \frac{1}{9} + \dots + \frac{1}{k^2} \rightarrow \frac{\pi^2}{6}$$

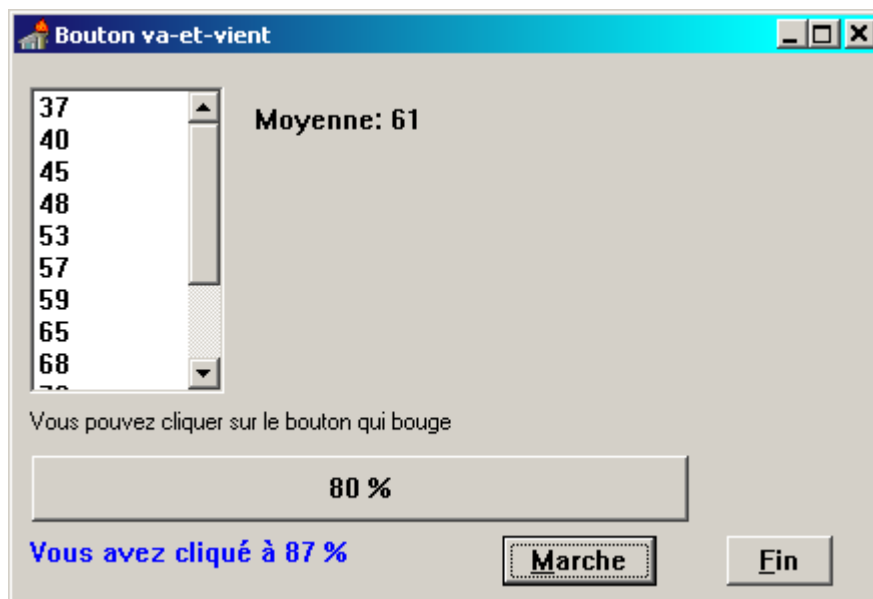
$$1 + \frac{1}{9} + \frac{1}{25} + \dots + \frac{1}{(2k+1)^2} \rightarrow \frac{\pi^2}{8}$$

Affichez la valeur de la constante pi définie dans Delphi, celles obtenues par les 3 approximations, ainsi que la différence entre chaque approximation et pi. L'utilisateur doit pouvoir choisir le nombre de termes calculés pour approximer pi.



Exercice 2.8:

Ecrire un programme qui se présente sous la forme suivante:



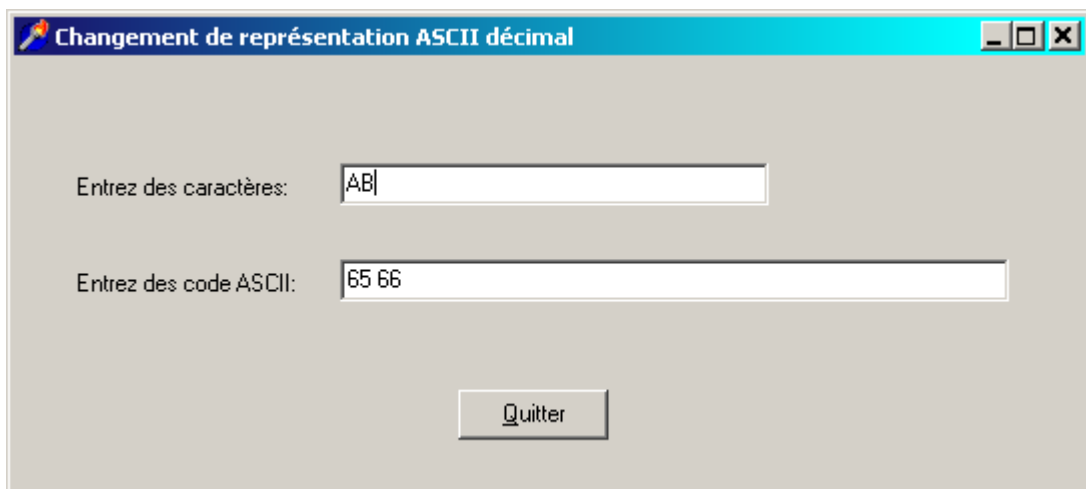
Le programme doit tenir compte des points suivants:

- un bouton voit sa largeur augmenter et diminuer continuellement
- un bouton marche/arrêt permet d'arrêter et de redémarrer le mouvement
- quand l'utilisateur clique sur le bouton qui bouge, l'indication du pourcentage au moment du clic est reportée sous le bouton. De plus la valeur (entière) du pourcentage est ajoutée dans une liste et la moyenne est calculée et affichée. La moyenne doit être entière.
- le bouton Fin termine le programme

Indications:

- il est conseillé d'utiliser un timer
- il ne faut pas stocker les valeurs autrement que dans la liste

Exercice 2.9:



Reproduire la fiche si-dessus.

L'utilisateur peut entrer des caractères.

A chaque changement de l'Edit contenant les caractères tapés par l'utilisateur, le programme affiche, dans le second Edit, les code des caractères séparés par un espace. Tenir compte également de la touche <BackSpace>.

Exercice 2.10:

Ecrire un programme simulant le fonctionnement d'une calculatrice simple (4 opérations).

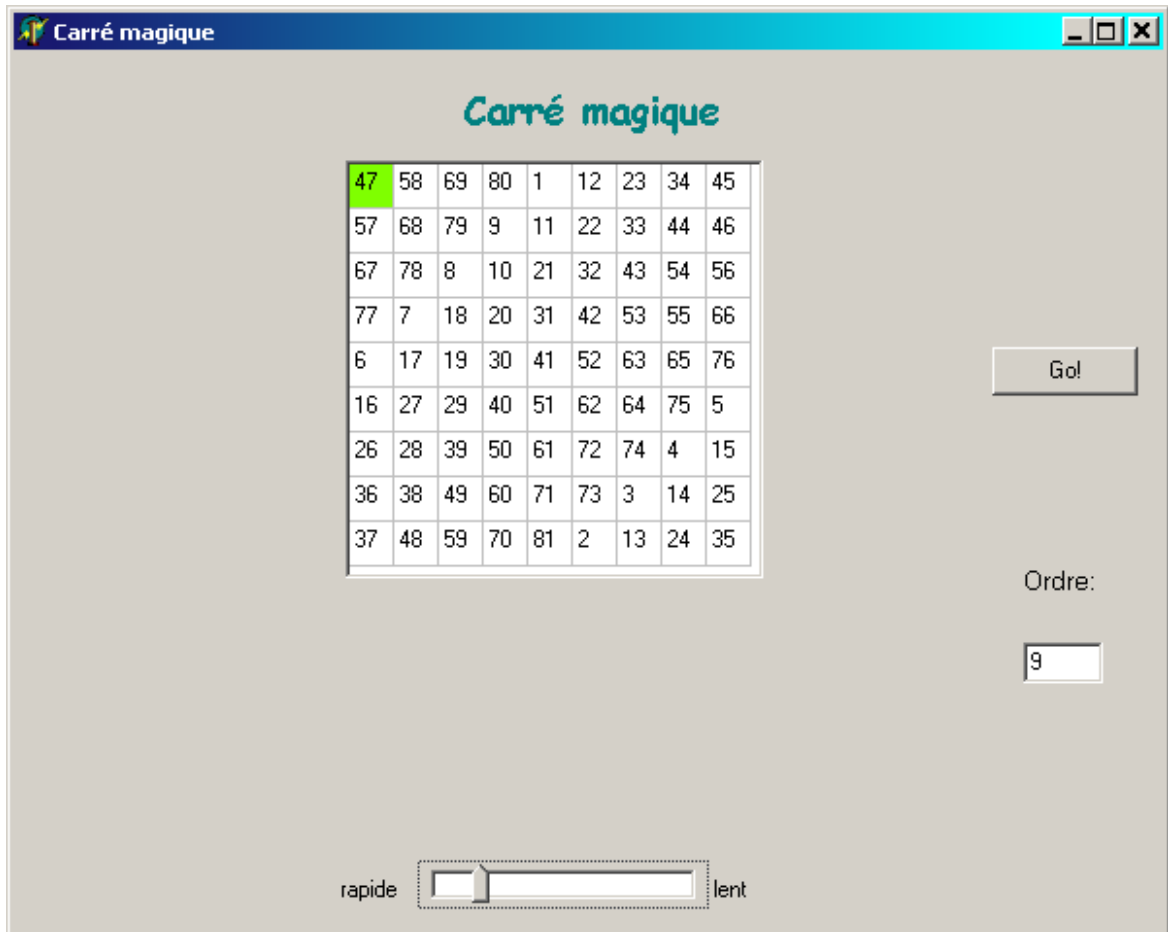


Exercice 2.11:

Ecrire un programme qui affiche, dans un StringGrid, un carré magique de dimension $n \times n$ (où n est impair et compris entre 3 et 15).

Dans un carré magique, la somme des nombres sur chaque ligne, chaque colonne et chaque (grande) diagonale doit être la même

Une méthode consiste à placer le 1 au milieu de la première ligne. Pour placer le nombre suivant, il faut se déplacer d'une case vers la droite et d'une case vers le haut. Lorsque l'on sort du carré, il faut rentrer par le côté opposé. Si on tombe sur une case occupée, on choisit alors la case qui est juste en dessous de la dernière case remplie.



Exercice 2.12:

Nombres premiers

Un nombre premier est un nombre entier divisible uniquement par 1 et par lui-même. Par exemple 1, 2, 3, 5, 7, ..., 17, 19, ..., 4549, 4561, 4567, ...

Comment déterminer si un nombre est premier ?

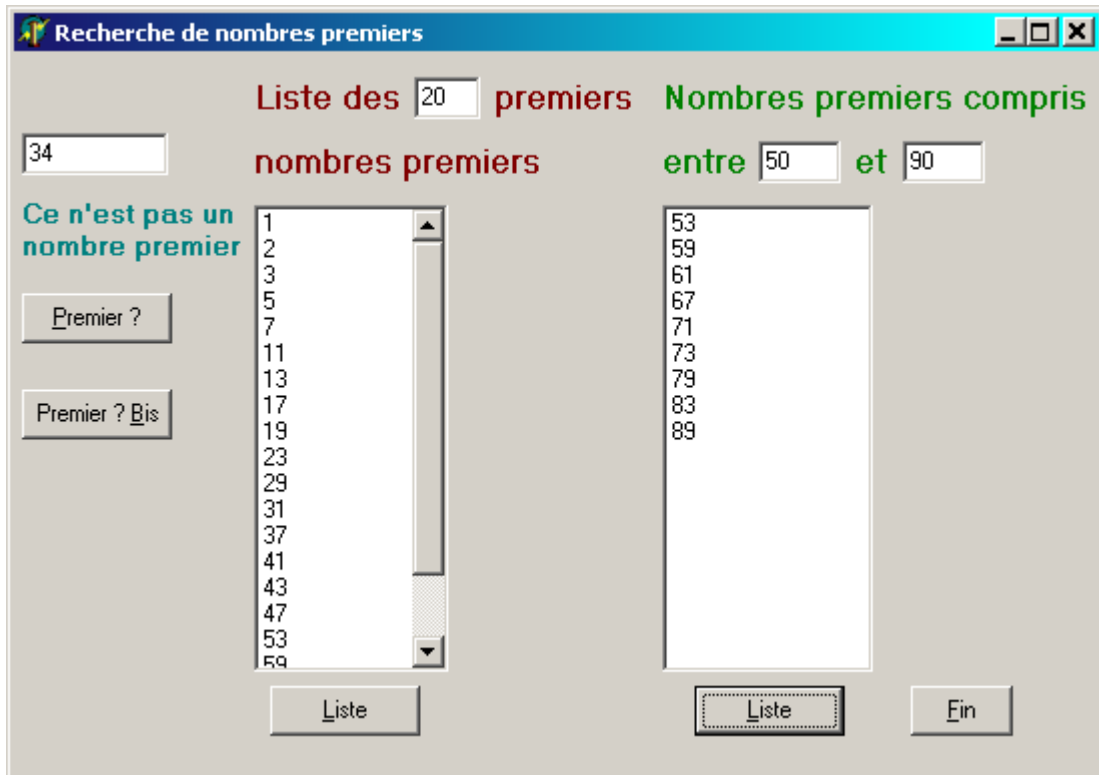
Il faut essayer de le diviser successivement par les nombres compris entre 2 et sa racine carrée. Si une des divisions est entière (n'a pas de reste), le nombre n'est pas premier. Si aucune des divisions n'est entière, le nombre est premier.

Ecrivez l'algorithme qui permet de déterminer si un nombre est premier.

Ecrivez un programme qui:

- Indique si un nombre donné par l'utilisateur est premier ou non
- Donne la liste des N premiers nombres premiers
- Donne la liste des nombres premiers compris entre X et Y

Voici l'aspect du programme:



Exercice 2.13:

Un petit peu de graphisme

Dans Delphi, on dessine sur la propriété **canvas** d'un composant. Nous allons utiliser un composant de type **TImage** (onglet Suppléments). Le crayon, **pen**, permet de dessiner (par défaut: couleur noire, épaisseur 1 pixel et style de trait continu).

Le système de coordonnées est "gradué" en **pixels**, l'origine (point (0, 0)) est située en haut à gauche. La largeur de l'image est indiquée par **Image.Width** et sa hauteur par **Image.Height** (en pixels).

Image.Canvas.MoveTo(X, Y: Integer);

Déplace le crayon au point (X,Y) sans laisser de trace. (Définit la position du crayon en (X, Y)).

Image.Canvas.LineTo(X, Y: Integer);

Dessine dans le canvas une ligne allant de la position en cours du crayon jusqu'au point de coordonnées spécifiées par X et Y, puis définit la position du crayon en (X, Y).

La ligne est dessinée en utilisant les propriétés de Pen.

Image.Canvas.Color := cbleu;

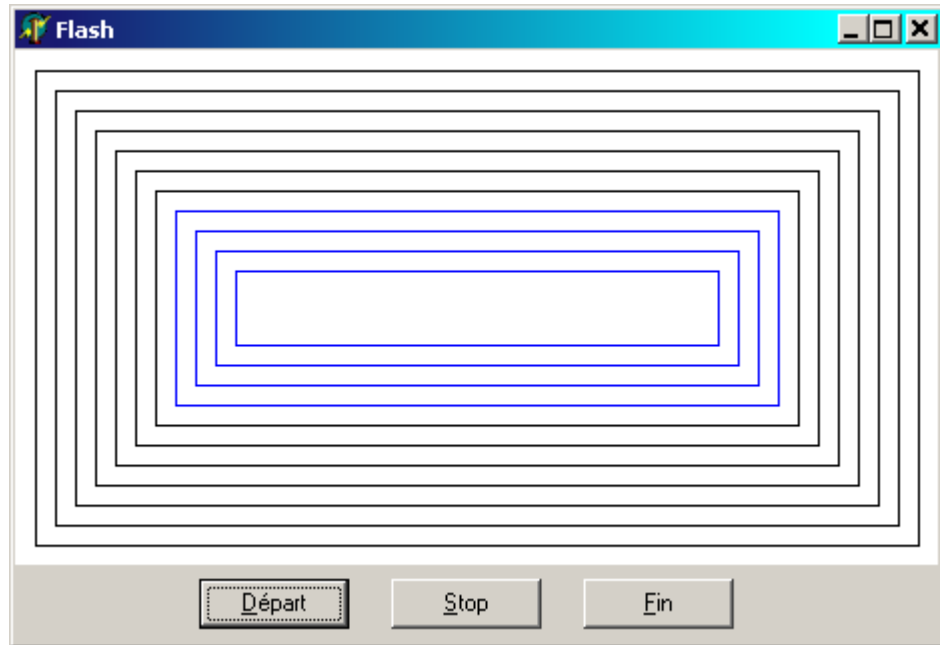
Initialise à bleu la couleur du crayon. A partir de cette instruction, le crayon dessinera en bleu.

Ecrire un programme qui, lorsque l'on appuie sur un bouton Départ, dessine des rectangles emboîtés (du plus grand au plus petit). L'utilisateur ne doit pas pouvoir changer la taille de la fiche.

Mais lorsque le programmeur change la taille de Form1, il ne doit pas avoir besoin de retoucher son code pour que le programme fonctionne correctement.

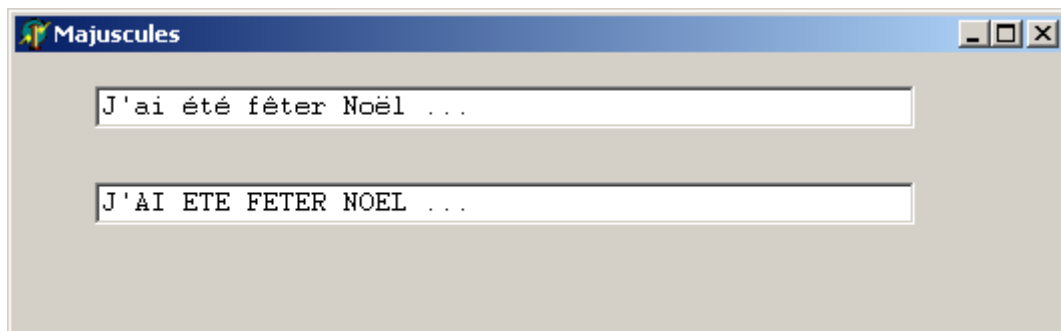
Amélioration 1: Lorsque tous les rectangles sont dessinés, les redessiner en gris en allant du plus petit au plus grand.

Amélioration 2: Le programme dessine tout le temps les rectangles (du plus grand au plus petit au plus grand...) jusqu'à ce que l'utilisateur appuie sur le bouton Fin.



Exercice 2.14:

Ce programme se compose de 2 Edits. Dans le premier, l'utilisateur peut taper des lettres qui sont transformées en majuscules et affichées au fur et à mesure dans le deuxième Edit.



La fonction **uppercase** retourne la majuscule correspondant à un caractère de l'alphabet. Pour les lettres accentuées, il faut analyser tous les cas et utiliser pour cela un **case of**.

Pour le traitement du BackSpace (#8), utiliser la procédure **delete** et la fonction **length** concernant les chaînes de caractères.

Servez-vous de l'aide de Delphi pour savoir comment utiliser les procédures et fonctions indiquées ci-dessus.

Exercice 2.15:

Ecrivez un programme qui fait avancer un trait (graphique) sur la fiche lorsque l'utilisateur appuie sur les flèches de déplacement.

Considérez l'événement **OnKeyDown**. Le paramètre **Key**, de type Word, peut prendre les valeurs suivantes qui nous intéressent:

- VK_RIGHT flèche vers la droite
- VK_DOWN flèche vers le bas
- VK_LEFT flèche vers la gauche
- VK_UP flèche vers le haut
- VK_HOME touche **Home** du pave numérique
- VK_PRIOR touche **Pg Up** du pave numérique
- VK_NEXT touche **Pg Down** du pave numérique
- VK_END touche **End** du pave numérique

Dans un premier temps, ne considérez que les flèches. Ensuite les touches du pavé numérique pour avancer de biais.

Toujours dans un premier temps, le trait se bloque si l'on veut aller plus loin que le bord de la fiche. Ensuite, si le trait sort d'un côté, il réapparaît de l'autre.

Voici l'aspect du programme:

