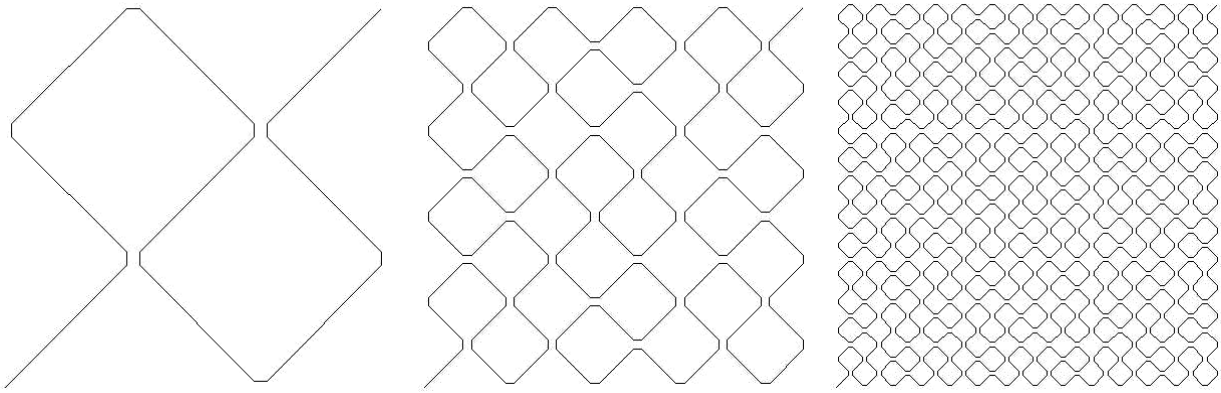
Lycée Frédéric Ozanam
Classes Préparatoires à l'ICAM et l'ISEN
Sup MPSI - PTSI

ALGORITHMIQUE et TURBO-PASCAL

Initiation à la programmation structurée



Année scolaire 2010-2011

Claude Lemaire
claude.lemaire@isen.fr

Sommaire

Chapitre 1 - Introduction
Compilateur et interpréteur ..... page 3
TURBO-PASCAL pour Windows : Un environnement intégré ..... page 4
Principes de la programmation structurée ..... page 7

Chapitre 2 - Les structures de l'algorithmique
Séquences ..... page 9
Conditions (Alternatives) ..... page 10
Boucles ..... page 11

Chapitre 3 - Introduction à la syntaxe Pascal
Variables, déclaration des variables ..... page 13
Structure d'un programme Pascal ..... page 13
Les instructions élémentaires : entrées, sorties, affectation ..... page 14
Tableaux ..... page 16
Enregistrements (Record) ..... page 17

Chapitre 4 - Fonctions
Introduction - Paramètres - Variables locales - Exemples..... page 18
Syntaxe Pascal ..... page 19

Chapitre 5 - Procédures
Introduction - Paramètres par valeur / par adresse - Variables locales..... page 21
Syntaxe Pascal ..... page 22
Démonstration de différents types de procédures d'échange ..... page 24

Travaux pratiques
Liste des TP année 2010-2011 ..... page 26
TP n°1 : Variables (Déclaration, Assignation) - Conditions ..... page 27
TP n°2 : Boucles - Suite de Fibonacci ..... page 28
TP n°3 : Boucles - Conditions ..... page 29
TP n°4 : Suites récurrentes ..... page 30
TP n°5 : Fonctions ..... page 32
TP n°6 : Tableaux ..... page 33
TP n°7 : Procédures ..... page 35
Sujet de synthèse n°1 : Nombre de dents d'un réducteur de vitesse à 2 étages.... page 37
Sujet de synthèse n°2 : Décharge d'un condensateur - Représentation graphique page 40
Sujet de synthèse n°3 : Graphique : suite de polygones emboîtés..... page 42
Annexe : Documentation de l'unité graphique WinGraph ..... page 44

Merci à Thierry Carette, Pierre Gervasi, Hélène Lemaire, Pascal Ricq, Roger Ringot et Michel Sénéchal pour leur amicale collaboration.

Compilateur et interpréteur

Un ordinateur est une machine susceptible d'exécuter certains traitements (programmes) sur des données codées en mémoire sous forme de *digits* (1 ou 0) ou d'*octets* (8 digits).

On peut distinguer 4 niveaux de fonctionnement :

1. **Le niveau LOGIQUE** : niveau le plus élémentaire.
   L'ordinateur apparaît comme un ensemble de circuits électroniques à réponse automatique (lorsque tel fil reçoit une tension haute, tel autre fil passe à la tension basse et tel autre fil passe à la tension haute). Le programmeur ne peut intervenir à ce niveau qui concerne uniquement le concepteur de l'ordinateur et de ses circuits intégrés
2. **Le langage machine**, à un niveau de complexité un peu plus grand.
   L'ordinateur apparaît comme susceptible d'exécuter des instructions contenues dans la mémoire. Ces instructions sont codées sous forme d'octets et c'est le rôle de *l'unité de commande* de traduire ces instructions au niveau logique pour commander à *l'unité logique*. Le programmeur peut intervenir à ce niveau en stockant les bons octets aux bons endroits (*adresses*), mais ce n'est pas facile.
   Les instructions exécutables à ce niveau sont :
   - Transfert d'octets d'une adresse à une autre
   - Opérations arithmétiques : addition, soustraction, multiplication ou division d'octets (ou éventuellement d'entiers représentés par 2 ou 4 octets)
   - Opérations logiques (ET, OU, NON, ...) sur les digits ou les octets.
   - Tests sur les octets (égalité, inégalité)
   - Sauts : Continuer la suite du programme à une autre adresse si une condition est vérifiée.
   - Autres (décalages, rotations).
3. A un niveau de complexité plus grand, le niveau de **l'assembleur**, l'ordinateur peut exécuter des opérations symboliques du type :
   - STO[24],12 : stocker le nombre 12 à l'adresse 24
   - ADD[48],[236] : additionner le contenu de l'adresse 236 à celui de l'adresse 48

   ou même utiliser des noms (*identificateurs*) pour désigner certaines adresses.
   Les instructions exécutables à ce niveau sont les mêmes qu'au niveau du langage machine, mais elles sont plus faciles à exprimer. Un programme (*un assembleur*) se chargera de traduire ces expressions symboliques en une suite d'octets qui pourra être comprise au niveau du langage machine.
4. Le dernier niveau de complexité est celui des **langages évolués** (BASIC, FORTRAN, PASCAL, C ou C++, ADA, MAPLE, JAVA...)
   A ce niveau, l'ordinateur pourra exécuter des instructions hautement symboliques, manipuler des données complexes autres que des octets (réels, tableaux, chaînes de caractères,...).
   En utilisant un *éditeur* (traitement de texte), le programmeur écrira ces instructions sous forme de texte (*code source*) et un programme sera chargé de traduire ce langage évolué en langage machine.
   Il y a 2 méthodes de traduction :
   - la " traduction simultanée " : le programme est traduit en langage machine au fur et à mesure de son exécution. Le programme de traduction est alors appelé un *interpréteur*.
   - L'autre solution est de traduire une fois pour toutes l'ensemble du programme avant de commencer à l'exécuter. Le programme de traduction est alors appelé un *compilateur*. A partir du *code source* (stocké sur le disque sous forme de fichier texte), le compilateur crée le *code exécutable* (stocké sur le disque sous forme de fichier de commandes .EXE ou .COM)

Le PASCAL est un langage *compilé*. Il comporte un éditeur de texte et un compilateur.

Le TURBO PASCAL est un *environnement intégré* permettant d'éditer un programme Pascal, le compiler et l'exécuter sans passer d'un programme à un autre, le *code source* et le *code exécutable* pouvant rester tous deux en mémoire.

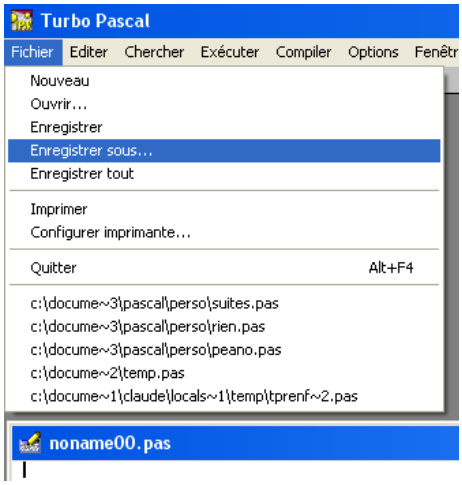
TURBO-PASCAL pour Windows : Un environnement intégré

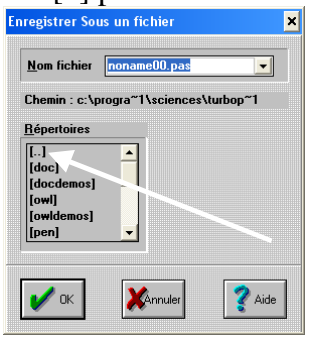
Turbo-Pascal Windows (TPW) comprend
- un éditeur de texte adapté au code source Pascal
- un compilateur
- un debugger

dans un environnement qui permet de passer de l'édition du code source à l'exécution du programme

1. Editeur

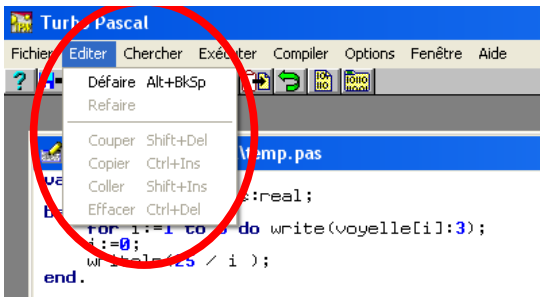
Quand on démarre Turbo-Pascal ou que l'on crée un nouveau fichier (noname...pas), la première chose à faire est de l'enregistrer, dans son dossier personnel (et surtout pas dans le dossier des programmes !).
Double-cliquer sur [..] pour sortir du dossier en cours.





Il est fortement recommandé de sauvegarder régulièrement
Le programme conserve l'avant-dernière version dans un fichier .bak (copie de sauvegarde)

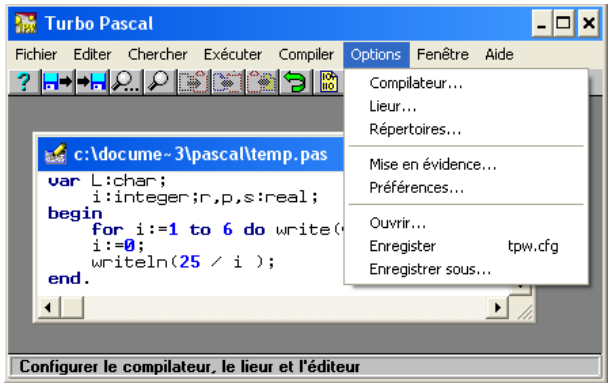
Attention : les raccoucis clavier sont ceux du windows standard et non pas ceux de Microsoft :



Utiliser plutôt le menu Edition

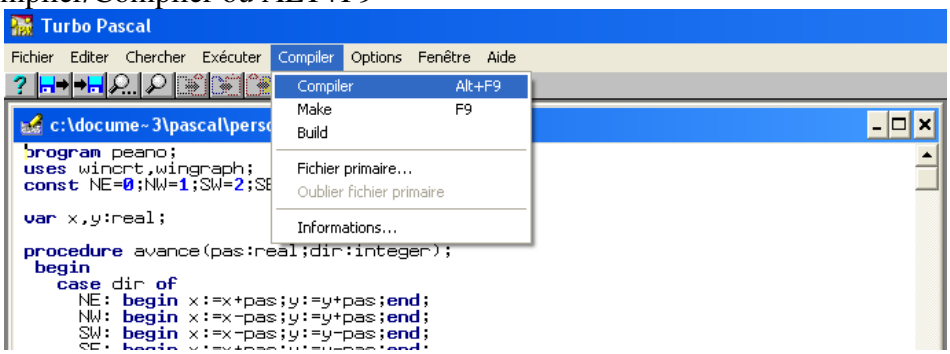
Utiliser les tabulations pour indenter le code (i.e. faire des retraits vers la droite). Lors du retour à la ligne, les indentations sont conservées. Touche BackSpace pour revenir sur la gauche.

Merci de ne pas modifier les options ni les préférences sur les machines qui sont mises à votre disposition, afin que chacun retrouve à chaque fois les mêmes options et préférences

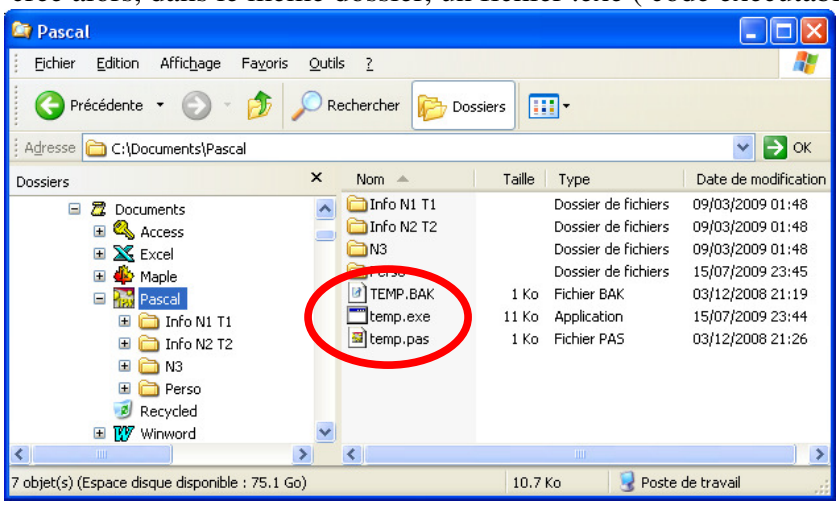


2. Compilateur

Une fois le code source rédigé et sauvegardé, il faut le compiler :
Menu Compiler/Compiler ou ALT+F9



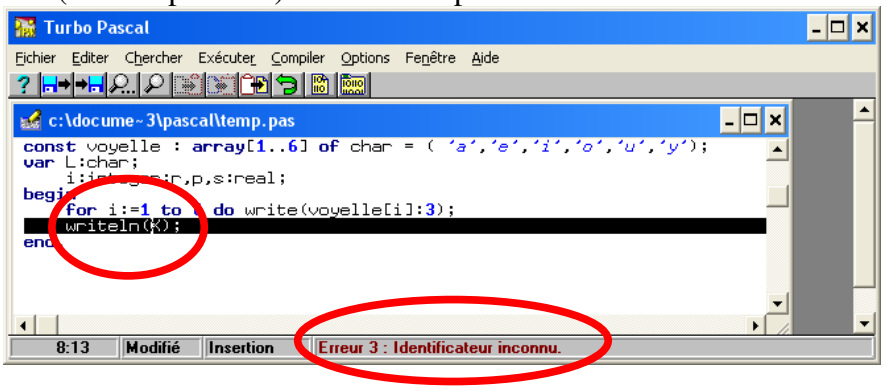
Le programme crée alors, dans le même dossier, un fichier .exe (code exécutable).



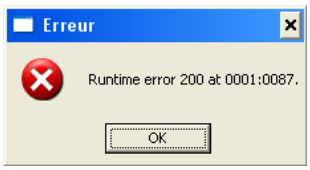
Ce fichier peut être exécuté en cliquant dans l'explorateur windows, ou (mieux) par le menu Exécuter/Exécuter ou CTRL+F9

3. Debugger

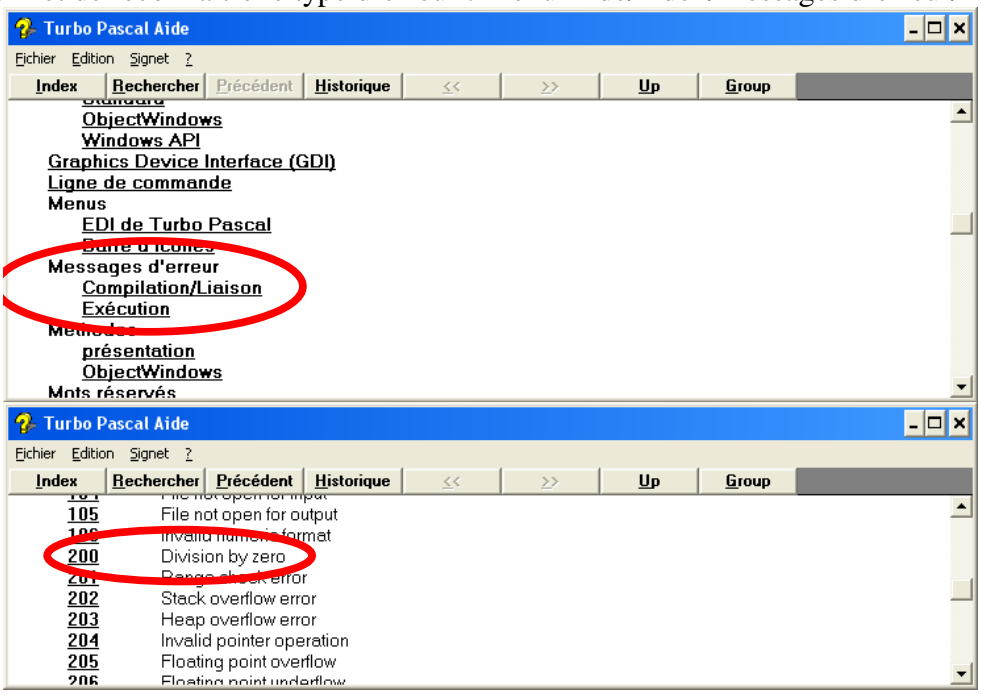
Lors de la compilation les erreurs sont repérées par le curseur clignotant sur la ligne surlignée.
Le code d'erreur (de compilation) et une description sont affichés en barre d'état



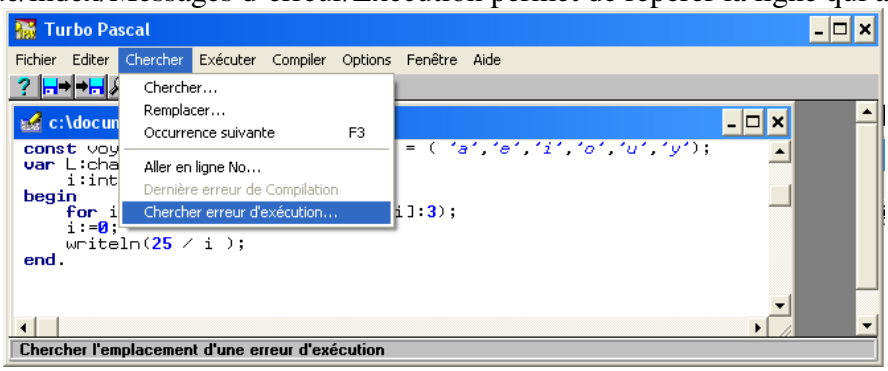
Lors de l'exécution une boîte de dialogue indique le code d'erreur (d'exécution) et l'adresse mémoire de l'erreur



L'aide permet de reconnaître le type d'erreur : Menu Aide/index/Messages d'erreur/Exécution

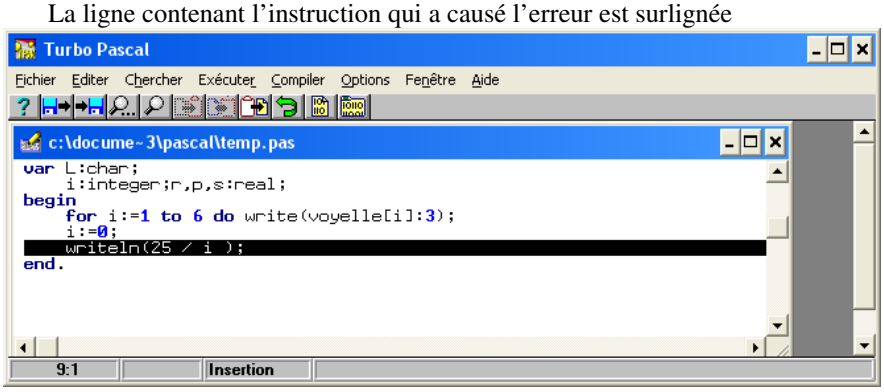


Le menu Aide/index/Messages d'erreur/Exécution permet de repérer la ligne qui a causé l'erreur



Taper l'adresse de l'erreur donnée dans la boîte de dialogue

La ligne contenant l'instruction qui a causé l'erreur est surlignée



(bien sûr ! division par 0 !!!)

On peut alors corriger le code source (sauvegarder) et exécuter à nouveau (le programme re-compile si le code source a été modifié)

Principes de la programmation structurée

Décomposition d'un problème

Elle comporte 4 phases successives :

1. Définir précisément le *cahier des charges* : ce que le programme devra faire.
2. Analyser le problème informatique :
   - Définir la *structure des données* : Quels types et quelles quantités d'informations doivent être traités ? Quelle est la meilleure manière de les gérer ?
   - Analyser l'organisation du programme : l'*algorithme*. La représentation la plus fructueuse de cette analyse est celle de l'*arbre programmatique*.
3. Traduire l'algorithme en langage évolué (ici PASCAL) : c'est le *codage*.
4. Compiler et exécuter le programme. Corriger à ce stade les erreurs éventuelles.

Remarque Le temps passé aux étapes 1. et 2. peut paraître astreignant mais est souvent rentable : Une analyse trop sommaire conduit souvent à des programmes qui ne " tournent " pas, ou (pire) qui " tournent mal " c'est à dire qu'ils donnent un résultat faux, ce dont on ne s'aperçoit pas forcément. On peut aussi dans ce cas passer un temps considérable à la mise au point et la correction des erreurs, et obtenir un programme confus, où peu de gens peuvent s'y retrouver, pas même parfois celui qui l'a écrit.

Algorithme

Un algorithme doit

- être *fini* (achevé après un nombre fini d'actions élémentaires)
- être *précis* (la machine n'a pas à choisir)
- être *effectif* (On pourrait le traiter " à la main " si on avait le temps)
- mentionner les *entrées* (saisie de données) et les *sorties* (affichage des résultats)

Le déroulement de tout algorithme peut se décrire avec les 3 *structures* suivantes :

1. ***SEQUENCE*** : suite d'instructions qui se succèdent (déroulement linéaire)
2. ***ALTERNATIVE*** : suivant le résultat d'un test on exécute une séquence ou une autre.
3. ***REPETITION (BOUCLES)*** : une instruction (ou une séquence) est répétée sous une certaine condition.

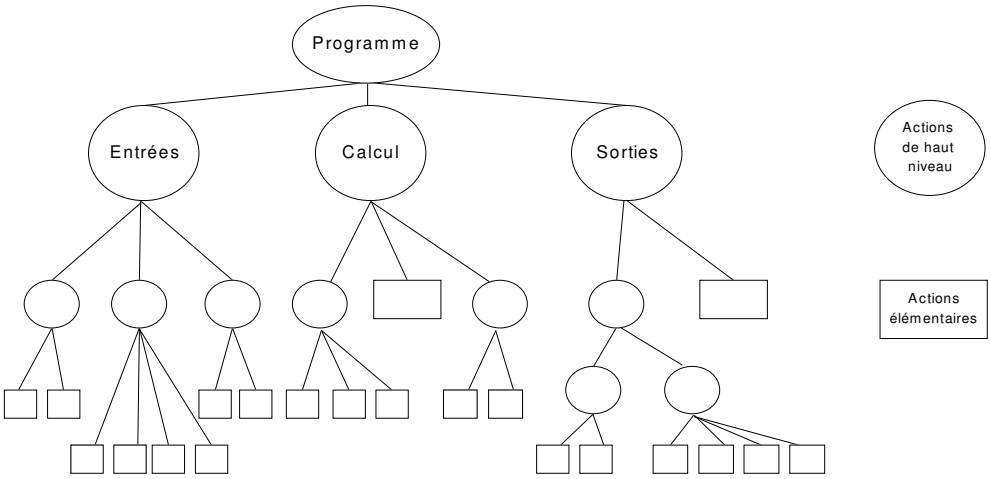
La représentation la plus fructueuse de cette analyse est celle de l'*arbre programmatique*.

- Les actions (instructions) élémentaires sont les feuilles de l'arbre.
- Ces actions élémentaires sont regroupées en une branche, et chaque branche est susceptible de posséder la même structure que l'arbre entier : *un sous-arbre est aussi un arbre*.
- L'écriture d'un arbre programmatique est donc la construction d'un arbre, de la racine vers les feuilles.

Cela correspond à une méthode d'analyse :

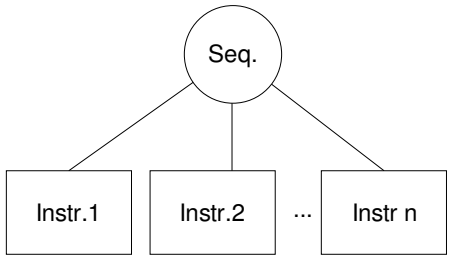
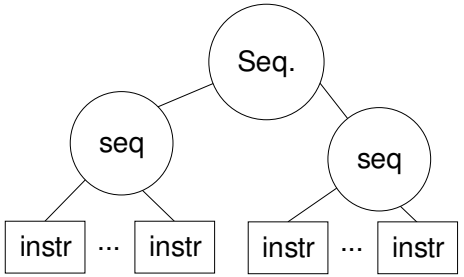
→ *descendante* : analyse par raffinements successifs.

→ *modulaire* : chaque branche peut être coupée du contexte et décrite séparément.



Les structures de base de l'algorithmique

1. Séquences

Arbre programmatique	Syntaxe Pascal	Syntaxe Maple	Syntaxe C
	**begin** Instruction1 ; Instruction2 ; . . . Instruction n1; **end** ;	Instruction1 ; Instruction2 ; . . . Instruction n ;	**{** Instruction1 ; Instruction2 ; . . . Instruction n ; **}**
	**begin** Instruction 1_1 ; Instruction 1_2 ; . . . Instruction 2_1 ; Instruction 2_2 ; . . . **end** ;	Instruction 1_1 ; Instruction 1_2 ; . . . Instruction 2_1 ; Instruction 2_2 ; . . .	**{** Instruction 1_1 ; Instruction 1_2 ; . . . Instruction 2_1 ; Instruction 2_2 ; . . . **}**

2. Alternatives

Arbre programmatique	Syntaxe Pascal	Syntaxe Maple	Syntaxe C
si Condition Instruction	**if** condition **then** Instruction **;**	**if** condition **then** Instruction **[;]** **fi ;** (**[;]** : *le point-virgule est optionnel*)	**if (condition)** Instruction **;**
si Condition Instruction_A Instruction_B	**if** condition **then** Instruction_A **else** Instruction_B **;** (*Pas de point-virgule avant* ***else***)	**if** condition **then** Instruction_A **[;]** **else** Instruction_B **[;]** **fi ;** (**[;]** : *le point-virgule est optionnel*)	**if (condition)** Instruction_A **;** **else** Instruction_B **;**

si
Condition
seq.
seq.
Instr_A1
Instr_A2
Instr_B1
Instr_B2
	if condition
	then
	begin
	Instruction_A 1 **;**
	Instruction_A2 **[;]**
	end
	else
	begin
	Instruction_B 1 **;**
	Instruction_B2 **[;]**
	end ;
	
	(**[;]** : *le point-virgule est optionnel*)
		if condition
		then
		Instruction_A 1 **;**
		Instruction_A2 **[;]**
		else
		Instruction_B 1 **;**
		Instruction_B2 **[;]**
		fi ;
		
		(**[;]** : *le point-virgule est optionnel*)
			if (condition)
			{
			Instruction_A 1 **;**
			Instruction_A2 **;**
			}
			else
			{
			Instruction_B 1 **;**
			Instruction_B2 **;**
			}

3. Boucles

Arbre programmatique	Syntaxe Pascal	Syntaxe Maple	Syntaxe C
pour Compteur fin debut Instruction	**for** compteur **:=** debut **to** fin **do** Instruction **;**	**for** compteur **from** debut **to** fin **do** Instruction **[;]** **od ;** (**[;]** : *le point-virgule est optionnel*)	**for (** compteur **=** debut **;** compteur **<=** fin **;** compteur++ **)** Instruction **;**
tant que condition Instruction	**while** condition **do** Instruction **;**	**while** condition **do** Instruction **[;]** **od ;** (**[;]** : *le point-virgule est optionnel*)	**while (** condition **)** Instruction **;**
répéter jusqu'à Instruction condition	**repeat** Instruction **[;]** **until** condition **;** (**[;]** : *le point-virgule est optionnel*)		**do** Instruction **;** **while (!** condition **) ;** (**!** *est la négation*)

Exemples de structures imbriquées

Arbre programmatique	Syntaxe Pascal	Syntaxe Maple	Syntaxe C

répéter
seq
condition
instr_1 instr_2 .. instr_n
	repeat
	Instr_1 ;
	Instr_2 ;
	. . .
	Instr_n ;
	until condition ;
			do
			{
			Instr_1 ;
			Instr_2 ;
			. . .
			Instr_n ;
			}
			while (**!** condition) ;

tant que
condition
si
test
instr_1 instr_2
	while condition **do**
	if test **then**
	Instr_1
	else
	Instr_2 ;
		while condition **do**
		if test
		then Instr_1 ;
		else Instr_2 ;
		fi ;
		od ;
			while (condition) **do**
			{ **if** (test)
			Instr_1 ;
			else
			Instr_2 ;
			}

pour
tant que
fin
Compteur
debut
condition
seq
instr_1 instr_2
	for compteur **:=** debut **to** fin
	do
	while condition **do**
	begin
	Instr_1 ;
	Instr_2 ;
	end ;
		for compteur **from** debut **to** fin
		do
		while condition
		do
		Instr_1 ;
		Instr_2 ;
		od;
		od ;
			for (compteur **=** debut ;
			compteur **<=** fin ;
			compteur**++**)
			{ **while** (condition)
			{
			Instr_1 ;
			Instr_2 ;
			}
			}

INTRODUCTION A LA SYNTAXE PASCAL

Remarque : L'éditeur PASCAL ne distingue pas les majuscules des minuscules, ignore les tabulations, les répétitions d'espaces et les retours à la ligne.
Ces mises en forme de texte sont donc facultatives.
Elles ont pour but la lisibilité du code source. Il y a donc intérêt à les utiliser rationnellement.

Au contraire, l'éditeur MAPLE et l'éditeur C distinguent les majuscules des minuscules.

Variables, déclaration des variables

Une *variable* est une place mémoire où est stockée une donnée sous forme d'octets.
L'*identificateur* de cette variable permet d'avoir accès à ces données sans être obligé de travailler sur les octets.
Les variables peuvent être de différents *types*, par exemple : entier (`INTEGER`), réel (`REAL`), booléen (`BOOLEAN`), caractère (`CHAR`), chaîne de caractères (`STRING`) ...
Déclarer une variable, c'est réserver une certaine place mémoire adaptée au type de la variable et lui associer un identificateur.
Syntaxe : `VAR` *`identificateur`* `:` *`type`* `;`

Exemples :
```
VAR i : INTEGER ;
VAR x1,x2 : REAL ;
VAR caractere : CHAR ;
```

Structure d'un programme

Syntaxe :
```
PROGRAM  nom_du_programme ;
USES wincrt ;                        { bibliothèque nécessaire pour read et write }

CONST identificateur = valeur ;      { déclaration ( éventuelle ) de(s) constante(s) }

TYPE nom_de_type = description ;     { déclaration ( éventuelle ) de(s) type(s) }

VAR  identificateur1 : type1 ;
     identificateur2 : type2 ;       { déclaration des variables }

BEGIN                                { début des instructions }
     instruction_1 ;
     instruction_2 ;

     ⋮
     instruction_n ;
END .                                { fin du programme }
```

Remarques :
- ▷ Les instructions sont séparées par ' `;` '
- ▷ Le retour à la ligne et les tabulations (*indentations*) sont facultatifs mais fortement recommandés.
- ▷ Le ' `END` ' final est suivi d'un point. Tout ce qui suivra sera ignoré par le compilateur.
- ▷ Ce qui est entre accolades { } est ignoré par le compilateur : Cela constitue un commentaire.

Instructions élémentaires :

Entrée de données

Syntaxe `READLN ( identificateur_de_variable ) ;`

Permet l'entrée au clavier d'une variable

Remarques :

Attention à ce que l'entrée corresponde bien au type de la variable !
Seul un curseur clignotant indique que l'ordinateur attend une entrée. Il est judicieux d'afficher auparavant un message pour indiquer ce qu'on attend.

langage C : `scanf ( "format_d_entree" , &identificateur_de_variable ) ;`

Affichage de données

Syntaxe `WRITE ( identificateur_de_variable ) ;`
Affiche à l'écran, à partir de la position courante du curseur, le contenu (la valeur) de la variable

`WRITE ( donnee_1 , donnee_2 , … , donnee_n ) ;`
Affiche à l'écran les données (contenu de variables, valeurs, ou textes) sur une même ligne, sans espace.

`WRITE ( identificateur_de_variable_entière : 5 ) ;`
Affiche à l'écran le contenu de la variable en utilisant 5 caractères (éventuellement espaces avant)

`WRITE ( identificateur_de_variable_réelle : 8 : 3 ) ;`
Affiche à l'écran le contenu de la variable avec 3 chiffres après la virgule,
en utilisant 8 caractères <u>en tout</u>, y compris le point décimal et les chiffres après la virgule.

`WRITELN ( . . . ) ;`
Même chose, mais passe à la ligne après l'affichage

Exemples
```
WRITELN ( 'la valeur de A à 2 décimales est ',A:6:2 ); { A est un real }
WRITE('n1 = ', n1:5 , 'n2 = ', n2:5);WRITELN;     { n1, n2 sont des integer }
```

langage C : `printf( "format_d_affichage" , identificateur_de_variable) ;`

Affectation

Syntaxe *identificateur_1* `:=` *identificateur_2* `;`
Affecte à la variable 1 le contenu de la variable 2
identificateur `:=` *expression* `;`
Affecte à la variable 1 le résultat de l'expression

Exemples où A, B et C sont de type `real` , i, j de type `integer` et S de type `string` (chaîne de caractères)

```
A := B ;          { assigne à A la valeur de B }
A := i ;          { assigne à A la valeur de i }
i := A ;   { donne une erreur de compilation : on ne peut assigner un real à un integer }
A := i/j ;        { assigne à A le quotient (de type real) de i par j }
i := i+1 ;        { i est incrémenté }
i := j div 10     { assigne à i le quotient (de type integer) de j par 10 ( éventuellement tronqué ) }
S := S+'.'        { ajoute le caractère '.' à la fin de la chaîne S }
```

Remarques : ▷ Ne pas confondre ' **:=** ' (affectation) avec ' **=** ' (comparaison des variables dans un test)

▷ Ne pas inverser les identificateurs ! ' `A := B` ' et ' `B := A` ' donnent des résultats différents !

langage C :
affectation : *identificateur* **=** *expression* ;
test d'égalité : *expression_1* **==** *expression_2* ;

langage Maple :
affectation : *identificateur* **:=** *expression* ;

Les tableaux

Un tableau est une variable contenant un nombre donné de variables d'un même type.
Attention : en Pascal, la taille d'un tableau est fixe : elle doit être une constante.
On déclare une telle variable avec le type **ARRAY**

Déclaration d'une variable de type tableau:
exemples :

- Un tableau A de 10 entiers numérotés de 1 à 10 `var A : array[1..10] of integer;`
- Un tableau B de 20 réels numérotés de -3 à 16 `var B : array[-3..16] of real;`

On peut aussi (et c'est préférable) déclarer un nouveau type que l'on pourra utiliser pour déclarer des variables :

Déclaration d'une constante :
`Const NomDeConstante = Valeur;`
exemple `const n = 5;`

Déclaration d'un type :
`type NomDeType = Description;`
exemple `type tab=array[1..n] of integer;`

Autres exemples

- Un type pour des matrices 3×3 : un tableau de 3 tableaux de 3 réels (3 lignes de 3 réels)
  `type mat = array[1..3] of array[1..3] of real;`
  autre syntaxe équivalente : un tableau à 2 indices
  `type mat = array[1..3,1..3] of real;`
  `var M1, M2 : mat ;`
- Un type tableau de réels pour représenter des polynômes. Les indices vont de 0 à une constante DegreMax à déclarer (assigner) au préalable

```
const DegreMax = 5 ;
type polynome = array[0..DegreMax] of real;
var P1,P2 : polynome ;
var T : array[1..5] of polynome ;  { un tableau de polynômes }
```

Utilisation dans un programme
On peut utiliser le tableau comme un unique objet. Exemple : `P1 := P2 ;`
On peut utiliser un élément donné du tableau :
L'élément d'indice i d'un tableau `T` est désigné par `T[i]`. Exemple : `T[3]:=T[1];`

Exemples d'instructions, avec les déclarations précédentes et une variable `i` de type **integer;**

- `A[2]:= 1 ;` { On assigne des valeurs à des éléments d'un tableau }
  `A[3]:=A[2]+1;`
- `B[0]:=1.2345;` { Calcul, stockage dans un tableau et affichage d'une suite récurrente }
  `for i:=1 to 16 do begin B[i]:=sqrt(B[i-1]);`
  `writeln(i:3,B[i]:10:6); end;`
- `for i:=0 to DegreMax do P1[i]:=1;` { P1 est le polynôme $1+X+X^2+...+X^{DegreMax}$ }
  `T[1]:=P1;` { le premier polynôme du tableau T est P1 }
  `T[2][3] := 2.22;` { le coefficient de degré 3 du polynôme T[2] est fixé à 2.22 }

Les enregistrements (RECORD)

Un enregistrement (RECORD) est un type de donnée structuré (tout comme un tableau (ARRAY))
A la différence d'un tableau,

- Un enregistrement peut contenir des données de types différents
- Les données d'un enregistrement ne sont pas forcément indexées par des entiers (ou d'autres « scalaires »)

Exemple 1

Un complexe sera représenté par un enregistrement composé de 2 réels.
L'enregistrement est composé de 2 « champs » notés `Re` et `Im`.
La déclaration de type sera :

```
TYPE complex = RECORD
                    Re, Im : real ;
               END;
```

Des variables `Z1` et `Z2` de ce type seront déclarées par

```
VAR Z1, Z2: complex;
```

On pourra alors utiliser dans un programme :

∗ les variables `Z1` et `Z2`. Par exemple :

```
Z1 := Z2;
```

∗ les différents champs de ces variables par `<nom_de_variable>.<nom_de_champ>`
Exemple:

```
Z1.Re:=3;
Z1.Im:=-1;
Z1.Im:=Z1.Re+Z2.Im;
writeln(Z1.Re:5:2,' + i * ',Z1.Im:5:3);
```

Exemple 2

Une date sera représentée par un enregistrement composé de 2 entiers et un mot.
L'enregistrement est composé de 3 « champs » notés `Jour, Mois` et `Année`.
La déclaration de type pourra être :

```
TYPE date = RECORD
              Jour : 1..31 ; { intervalle de N }
              Mois : string ; { chaîne de caractères }
              Annee : 1900..2100; { entier dans un intervalle }
            END;
```

Des variables `avant` et `maintenant` de ce type seront déclarées par

```
VAR avant,maintenant : date;
```

On pourra alors utiliser dans un programme :

- les variables `avant` et `maintenant`. Par exemple :

```
avant:=maintenant;
```

- les différents champs de ces variables. Exemple:

```
avant.Jour:=15;
avant.Mois:='Octobre';
avant.Annee:=1995;       { 15/10/95 }
if (maintenant.Mois=avant.Mois)
 and(maintenant.Annee=avant.Annee)
then nb_jours:=maintenant.Jour-avant.Jour;
```

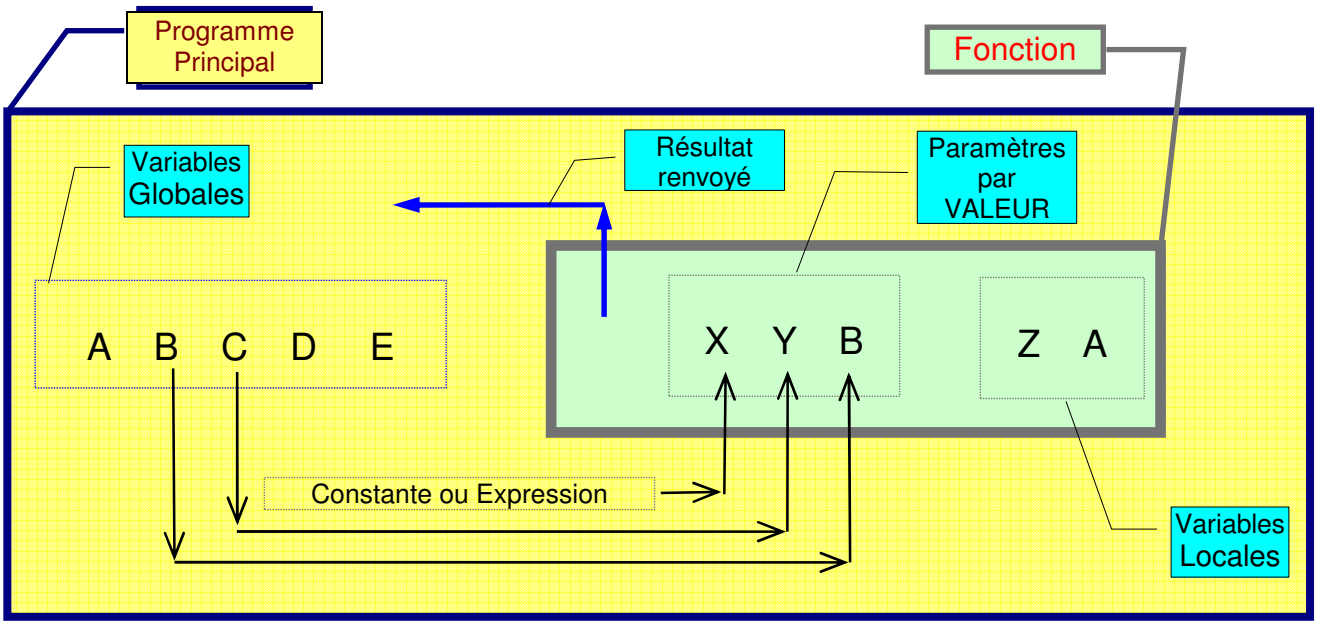
Les fonctions

Introduction : Une fonction est un sous-programme, défini séparément du « programme principal », et pouvant être appelé depuis ce programme principal pour **calculer une valeur** (d'un type "standard" : REAL, INTEGER, BOOELAN, CHAR, ou STRING) à partir de certaines données (les **paramètres par valeur**) fournies par le programme principal.

Questions à se poser

a/ Que doit calculer la fonction ?
b/ A partir de quelles données calcule-t-on le résultat ?
c/ Comment ? :

- Quel algorithme ?
- De quelles variables a-t-on besoin pour le calcul ? Ce seront des **variables locales**.



Exemples :

Plusieurs fonctions sont programmées dans Turbo-Pascal :

Des fonctions mathématiques : `ln`, `exp`, `sin`, `cos` (mais pas `tan`) `arctan` (mais pas `arcsin`)

Ces fonctions ont un paramètre de type `REAL` et donnent un résultat de type `REAL`.

Partie entière d'un paramètre de type `REAL`:

`int` donne un résultat de type `REAL`,

`trunc` et `round` donnent un résultat de type `INTEGER` (ou `LONGINT`).

La fonction `odd` pour tester la parité d'un entier : un paramètre de type `INTEGER`, résultat de type `BOOLEAN`. (vrai si le paramètre est impair)

Des fonctions sur les chaînes de caractères :

`length` (calcule la longueur d'une chaîne)

Un paramètre de type `STRING`, résultat de type `INTEGER`

`chr(n)` donne le caractère dont le n° ASCII est `n`,

`ord(c)` donne le numéro ASCII du caractère `c`.

Exemples : `chr(65)` donne `'A'`, `ord('B')` donne `66`

Syntaxe Pascal

Déclaration d'une fonction

```
FUNCTION NomDeFonction(DeclarationDesParametres):TypeDuResultat;
```

où `DeclarationDesParametres` est une suite de termes de la forme
`NomDeParametre:TypeDeParametre` séparés par des `';'`
et `TypeDuResultat` est l'un des types prédéfinis :
`INTEGER, REAL, BOOLEAN, CHAR, STRING, ...`

remarque : pour renvoyer la valeur calculée au programme principal, il faut l'instruction suivante:
`NomDeFonction:=ValeurDuResultat;`

exemple :

```
function factorielle(n:integer):integer;
    var k,produit:integer;
      begin
            produit:=1;
            for k := 1 to n do produit:=produit*k;
            factorielle:=produit;
      end;
```

Appel d'une fonction dans le programme principal

On utilise une fonction dans une expression ou un appel de procédure, exactement comme une variable, mais en indiquant les paramètres entre parenthèses et séparés par des virgules.

exemple :

Si on a déclaré la fonction `factorielle` comme ci-dessus, et si `k` est de type `integer`

```
k := factorielle(3);
for i:=1 to 7 do writeln(i,factorielle(i):6);
```

Structure générale d'un programme utilisant des fonctions

`PROGRAM NomDuProgramme ;`	
`CONST ... = ... ;`	Constantes Globales
`TYPE ... = ... ;`	Types globaux
`FUNCTION Funct(Parametres):TypeRes;`	Declaration de fonction globale
`CONST ... = ... ;`	Constantes locales à la fonction Funct
`TYPE ... = ... ;`	Types locaux à la fonction Funct
`VAR ... : ... ;`	Variables locales à la fonction Funct
`BEGIN`	
`...`	Corps de la fonction Func1
`...`	
`Funct := ...`	Pour renvoyer la valeur calculée au programme principal,
`END;`	
`VAR ... : ... ;`	Variables globales
`BEGIN`	
`...`	Corps du programme principal
`...`	
`END.`	Fin du programme

Exemple : Une fonction `pgcd` pour calculer le pgcd de 2 entiers.

```
function pgcd(a,b:integer):integer;
var r:integer;
begin
     r:=b;
     while r<>0 do begin r:=a mod b;
                         a:=b;
                         b:=r;
                   end;
     pgcd:=a;
  end;
```

- pgcd est le nom choisi pour la fonction
- les paramètres par valeur sont les entiers `a` et `b` dont on calcule le pgcd : paramètres formels (ie variables muettes)
- le résultat `(pgcd)` est un entier
- On a besoin d'une variable locale de type entier
- C'est le "corps" de la fonction : les instructions qui servent à calculer le résultat.
- à la fin de la boucle, la variable a contient le résultat cherché. La dernière instruction assigne ce résultat à la fonction. C'est cette valeur qui sera renvoyée au programme principal

Exemple d'utilisation de cette fonction dans un programme

```
var a,b, n : integer
begin
  readln(a);readln(b);readln(n);
  writeln( pgcd(a,b) );
  writeln( pgcd(a,n) );
  if pgcd(b,n) = 1 then writeln('b et
         n sont premiers entre eux');
  writeln( pgcd( n, pgcd(a,b)) );
end.
```

- Les variables du programme principal (variables globales)
- On affiche le pgcd de `a` et `b`
- On affiche le pgcd de `a` et `n`
- On utilise ce pgcd dans un test
- On affiche le pgcd de `n` et du pgcd de `a` et `b`

Les procédures

Introduction : Une procédure (ou « module ») est un sous-programme, défini séparément du « programme principal », et pouvant être exécuté (« appelé ») depuis ce programme principal.

Question à se poser

a/ Que fait la procédure ?
b/ Sur quels objets (« données ») travaille la procédure ?
c/ Comment ? (quel algorithme ?)

pour le b/ on distingue :

- les variables locales
- les variables globales
- les paramètres
  - paramètres « par valeur » (en entrée) (« IN »)
  - paramètres « par adresse » (en entrée/sortie) (« INOUT »)

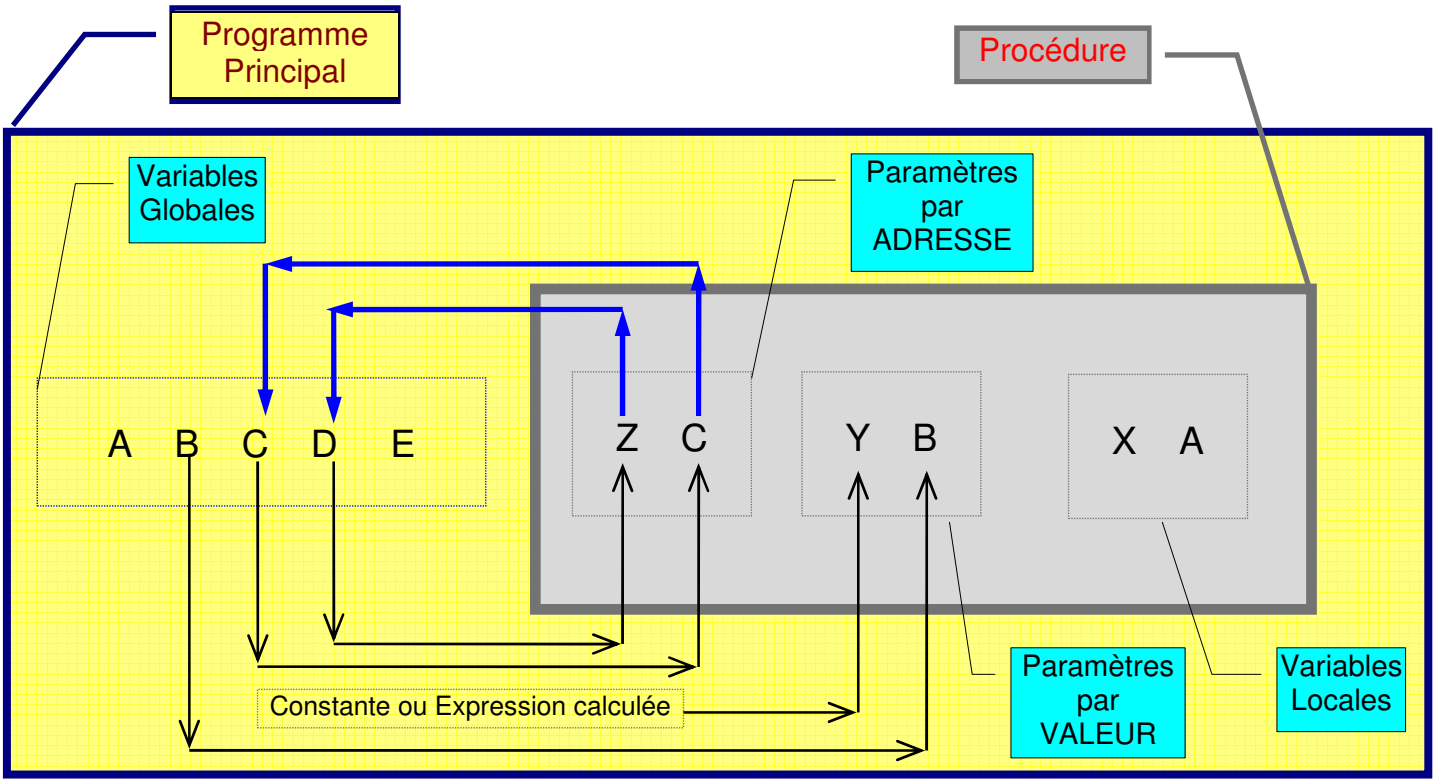
Paramètres par valeur, paramètres par adresse

Un paramètre par valeur reçoit une valeur du programme principal, soit le contenu d'une variable globale, soit le résultat d'une expression.
A la fin de la procédure, la variable globale n'est pas modifiée.

Un paramètre par adresse reçoit une valeur du programme principal, contenu d'une variable globale. Toute modification du paramètre par adresse modifie en même temps la variable globale associée.

Si la procédure a pour objet de modifier la variable associée à un paramètre, il faut déclarer ce paramètre par adresse.

Si la variable associée à un paramètre n'a pas à être modifiée, il faut (sauf exception) déclarer ce paramètre par valeur.



Déclaration d'une procédure

```
PROCEDURE NomDeProcedure( DeclarationDesParametres );
```

où `DeclarationDesParametres` est une suite de termes de la forme
`NomDeParametre:TypeDeParametre` (pour des paramètres par valeur)
VAR `NomDeParametre:TypeDeParametre` (pour des paramètres par adresse)
séparés par des `';'`

Appel d'une procédure

`NomDeProcedure( PassageDesParametres );`
où `PassageDesParametres` est de la forme
`Parametre1, Parametre2, ... ,Parametre_n`
(`Parametre1, Parametre2, ... ,Parametre_n` étant les paramètres correspondant à ceux de la déclaration de procédure, dans le même ordre, mais sans préciser le type et séparés par des virgules)

Exemple : Une procédure `norme` pour calculer le complexe $Z = \frac{z}{|z|}$, à partir d'un complexe z .

Dans la procédure, on notera `x,y` les composantes de z et `u,v` celles de Z .

```
PROCEDURE norme( x,y : real ;
                 var u,v : real );
var module :real ;
begin
      module := sqrt(x*x+y*y) ;
      u := x/module ;
      v := y/module ;
end;
```

- `norme` est le nom choisi pour la procédure
- `x,y` (composantes de z) sont des paramètres par valeur : ils n'ont pas à être modifiés.
- `u,v` sont des paramètres par adresse : c'est le but de la procédure de modifier ces valeurs.
- On a besoin d'une variable locale
- C'est le "corps" de la procédure

Exemple d'utilisation de cette procédure dans un programme

```
var a,b,u,v : real
begin
  readln(a); readln(b);
  norme(a,b,u,v );
  writeln( u:8:3 , u:8;3 );
  norme(1,2,u,v );
  writeln( u:8:3 , u:8;3 );
  norme(a,b,a,b );
  norme(a,b,a,b );
end.
```

- Les variables du programme principal (variables globales)
- On exécute la procédure, ce qui assigne `u` et `v` pour qu'on ait $u + i\,v = \frac{a + i\,b}{\sqrt{a^2 + b^2}}$
- On affiche `u` et `v` avec 3 chiffres après la virgule
- On exécute la procédure, ce qui assigne `u` et `v`
- On aura `0.447` ($= \frac{1}{\sqrt{5}}$) et `0.894` ($= \frac{2}{\sqrt{5}}$)
- On exécute la procédure, ce qui modifie `a` et `b` $a + i\,b$ a changé et a maintenant un module = 1
- On exécute la procédure, ce qui modifie `a` et `b` $a + i\,b$ a changé et a maintenant un module = 1

Structure générale d'un programme

```
PROGRAM NomDuProgramme ;
   CONST ... = ... ;                          Constantes Globales
   TYPE ... = ... ;                           Types globaux
   VAR ... : ... ;                            Variables globales

   PROCEDURE Proc1(Parametres);               Déclaration de procédure globale
      CONST ... = ... ;                       Constantes locales à la procédure Proc1
      TYPE ... = ... ;                        Types locaux à la procédure Proc1
      VAR ... : ... ;                         Variables locales à la procédure Proc1
      BEGIN
         ...                                  Corps de la procédure Proc1
         ...
      END;

   PROCEDURE Proc2(Parametres);               Déclaration de procédure globale
      CONST ... = ... ;                       Constantes locales à la procédure Proc2
      TYPE ... = ... ;                        Types locaux à la procédure Proc2
      VAR ... : ... ;                         Variables locales à la procédure Proc2
      PROCEDURE    SProc2(Parametres);        Déclaration de procédure locale à Proc2
         CONST ... = ... ;                    Constantes locales à la procédure SProc2
         TYPE ... = ... ;                     Types locaux à la procédure SProc2
         VAR ... : ... ;                      Variables locales à la procédure SProc2
         BEGIN
            ...                               Corps de la procédure SProc2
            ...
         END;
      BEGIN
         ...                                  Corps de la procédure Proc2
         ...
      END;

   FUNCTION Func1(Parametres):TypeRes;        Declaration de fonction globale
      CONST ... = ... ;                       Constantes locales à la fonction Func1
      TYPE ... = ... ;                        Types locaux à la fonction Func1
      VAR ... : ... ;                         Variables locales à la fonction Func1
      BEGIN
         ...                                  Corps de la fonction Func1
         ...
      END;

   BEGIN
      ...                                     Corps du programme principal
      ...
   END.                                       Fin du programme
```

Démonstration de différents types de procédures d'échange : le code Pascal

```
PROGRAM Demonstration_differents_types_de_procedures_d_echange;
USES wincrt;
VAR a,b,c,d,e:integer;

procedure echange1;
   { échange le contenu des variables a et b }
   { version 1 avec variables globales seulement }
  begin
    c:=a;
    a:=b;
    b:=c;
  end;

procedure echange2;
   { échange le contenu des variables a et b }
   { version 2 avec deux variables globales et une locale }
  VAR temp:integer;
  begin
    temp:=a;
    a:=b;
    b:=temp;
  end;

procedure echange3(a,x:integer);
   { échange le contenu des paramètres a et x }
   { version 3 avec variable globale et paramètres par VALEUR }
  begin
    c:=a;
    a:=x;
    x:=c;
  end;

procedure echange4(a,x:integer);
   { échange le contenu des paramètres a et x }
   { version 4 avec variable locale et paramètres par VALEUR }
  VAR temp:integer;
  begin
    temp:=a;
    a:=x;
    x:=temp;
  end;

procedure echange5(VAR a,x:integer);
   { échange le contenu des paramètres a et x }
   { version 5 avec variable locale et paramètres par ADRESSE }
  VAR temp:integer;
  begin
    temp:=a;
    a:=x;
    x:=temp;
  end;

begin                      { programme principal }
  writeln;writeln(' version 1 ');
    a:=3;b:=4;c:=5; write('a=',a,'  b=',b,'  c=',c);
    echange1; write(' --> On échange a et b -->  '); writeln('a=',a,'  b=',b,'  c=',c);

  writeln;writeln(' version 2 ');
    a:=3;b:=4;c:=5; write('a=',a,'  b=',b,'  c=',c);
    echange2; write(' --> On échange a et b -->  '); writeln('a=',a,'  b=',b,'  c=',c);

  writeln;writeln(' version 3 ');
    a:=3;b:=4;c:=5; write('a=',a,'  b=',b,'  c=',c);
    echange3(a,b); write(' --> On échange a et b -->  '); writeln('a=',a,'  b=',b,'  c=',c);

  writeln;writeln(' version 4 ');
    a:=3;b:=4;c:=5; write('a=',a,'  b=',b,'  c=',c);
    echange4(a,b); write(' --> On échange a et b -->  '); writeln('a=',a,'  b=',b,'  c=',c);
    a:=3;b:=4;c:=5; write('a=',a,'  b=',b,'  c=',c);
    echange4(b,c); write(' --> On échange b et c -->  '); writeln('a=',a,'  b=',b,'  c=',c);

  writeln;writeln(' version 5 ');
    a:=3;b:=4;c:=5; write('a=',a,'  b=',b,'  c=',c);
    echange5(a,b); write(' --> On échange a et b -->  '); writeln('a=',a,'  b=',b,'  c=',c);
    a:=3;b:=4;c:=5; write('a=',a,'  b=',b,'  c=',c);
    echange5(b,c); write(' --> On échange b et c -->  '); writeln('a=',a,'  b=',b,'  c=',c);

end.
```

Démonstration de différents types de procédures d'échange : Résultats

```
(Inactive D:\DOCUME~1\PASCAL\INFON1~1\COURS\ECHANGES...

 version 1
a=3  b=4  c=5 --> On échange a et b -->  a=4  b=3  c=3

 version 2
a=3  b=4  c=5 --> On échange a et b -->  a=4  b=3  c=5

 version 3
a=3  b=4  c=5 --> On échange a et b -->  a=3  b=4  c=3

 version 4
a=3  b=4  c=5 --> On échange a et b -->  a=3  b=4  c=5
a=3  b=4  c=5 --> On échange b et c -->  a=3  b=4  c=5

 version 5
a=3  b=4  c=5 --> On échange a et b -->  a=4  b=3  c=5
a=3  b=4  c=5 --> On échange b et c -->  a=3  b=5  c=4
```

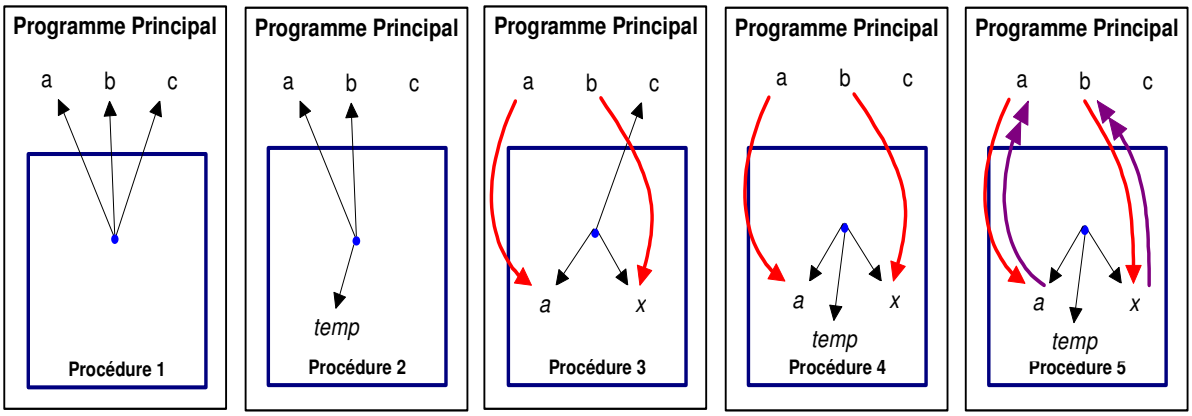
Procédure 1 : Il n'y a pas de paramètres ni de variable locale. On n'agit que sur les variables globales.
a et b sont bien échangés mais c est modifié alors que ce n'était pas souhaité.
Autre inconvénient : Cette procédure ne permet que d'échanger les variables a et b.
Pour échanger a et d il faudrait une autre procédure. Cette procédure n'est pas assez générale.

Procédure 2 : Il n'y a pas de paramètres mais une variable locale.
On n'a donc pas l'effet indésirable de modifier c, mais l'autre inconvénient demeure :
On ne peut échanger que les variables a et b. Elle n'est pas assez générale.

Procédure 3 : On a deux paramètres par valeur. Les variables associées ne sont donc pas modifiées.
De plus la variable globale c est modifiée.
Bref cette procédure ne fait pas ce qu'elle devrait faire et fait ce qu'elle ne doit pas !

Procédure 4 : On a deux paramètres par valeur et une variable locale.
Les variables associées aux paramètres par valeur ne sont donc pas modifiées.
Bref cette procédure ne fait pas ce qu'elle ne doit pas faire mais ne fait pas non plus ce qu'elle doit faire !

Procédure 5 : On a deux paramètres par adresse et une variable locale.
Les variables associées aux paramètres par valeur sont donc modifiées par la procédure.
Et les autres variables globales ne sont pas modifiées.
L'utilisation de paramètres par adresse permet d'utiliser le même procédure pour échanger
le contenu de n'importe que couple de 2 variables (exemple b et c) et ce, sans effet indésirable



Morale :

- Pas de variables globales dans une procédure (ou une fonction), mais :
  - des paramètres s'il s'agit de données à échanger entre la procédure et le programme principal
  - des variables locales si ce sont des variables pour des calculs intermédiaires.
- Choisir le bon type de paramètre :
  - par valeur si la variable associée n'a pas à être modifiée
  - par adresse si c'est l'objet de la procédure de modifier la variable associée.