Les Pointeurs en Pascal

Université de Toulouse II

DEUG MASS

Année 2004-2005

Guillaume Lussier

< guillaume.lussier@laas.fr >

Plan du cours sur les pointeurs

- approche des pointeurs, notions de zones mémoires *(p3)*
- présentation des pointeurs *(p6)*
  - déclaration, allocation
  - utilisation sans allocation
  - la valeur `nil`
  - comparaison et affectation de pointeurs

- structures récursives *(p15)*
  - présentation, la liste simplement chaînée
  - insertion, suppression dans une liste chaînée
- conclusion *(p23)*

Approche de la notion de pointeur

Jusqu'à présent la représentation d'un groupe de données de même nature (même type) n'est possible qu'au moyen des **tableaux**. Or leur facilité d'utilisation peut être fortement réduite lorsque:

a) la structure séquentielle d'un tableau ***ne reflète pas l'organisation globale des données*** comme par exemple : les liaisons ferroviaires ou routières entre les différentes gares ou villes d'une région; tout arbre de calcul.
En effet un tableau peut difficilement refléter les successeurs ou prédécesseurs d'une donnée, les relations existant entre certaines données.

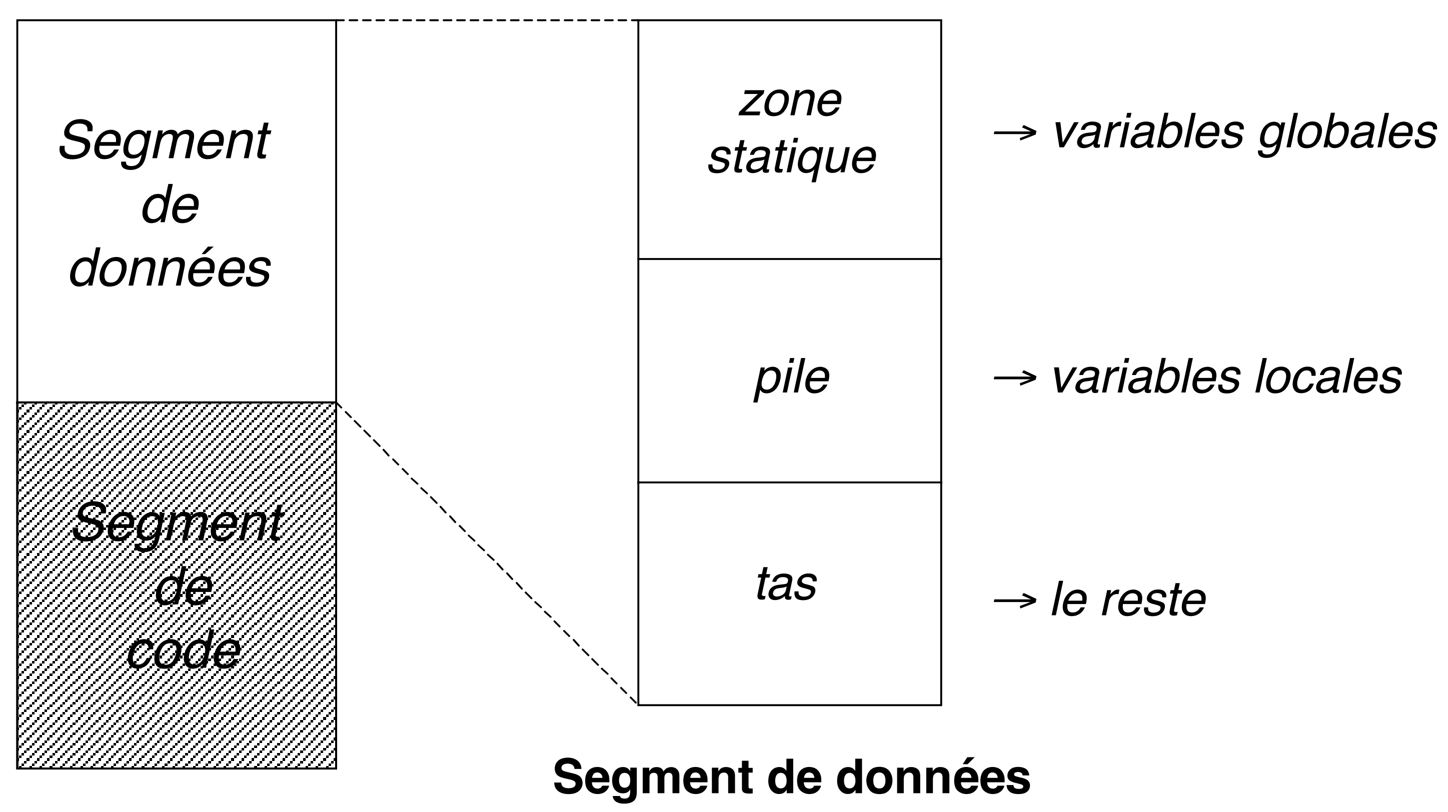
b) le nombre maximal d'éléments du tableau n'est ***pas connu à la compilation*** (ne peut pas être fixé par le programmeur), à savoir

- si ce nombre est choisi trop petit le programme ne pourra représenter et traiter toutes les données;
- s'il est pris (beaucoup) trop grand le gaspillage de mémoire peut pénaliser non seulement le fonctionnement du programme mais aussi celui du système informatique entier.

On a besoin d'une structure plus souple pour manipuler les données, pour cela nous allons étudier un autre moyen d'y accéder : **les pointeurs**.

Notion de zones mémoires (1)

- zones mémoires utilisées par l'exécution d'un programme



Notion de zones mémoires (2)

Un programme qui s’exécute utilise au moins deux zones mémoires, appelées segments :

- *le segment de code :* contient les instructions du programme;
- *le segment de données :* contient les données utilisées par le programme.

Le segment de données est découpé en trois parties :

- *la zone statique :* contient les variables statiques, c’est-à-dire les variables ayant une durée de vie égale à celle du programme : c’est le cas des variables globales.

- *la pile :* sert à gérer les éléments du programmes dont la durée de vie est limitée à un bloc : c’est le cas des paramètres des appels de fonctions/procédures, des variables locales. Elle sert également à renvoyer les résultats des fonctions.

- *le tas :* tout ce qui reste... Un programme ne peut accéder directement aux emplacements du tas. Par contre, on peut allouer des emplacements dans le tas et les référencer ensuite à l’aide de pointeurs. La gestion du tas est à la charge du programmeur.

Pointeurs : déclaration et allocation (1)

Un pointeur permet de représenter des données complexes, de modifier le contenu d'une variable, de travailler directement sur le contenu de la mémoire en utilisant le principe de l'*allocation dynamique*.

Un pointeur est ni plus ni moins un moyen d'*accéder indirectement* à une variable : au lieu d'accéder à celle-ci, on passe par le pointeur qui nous dirige vers celle-ci.

Exemple :

```
program pointeur;
    type T_Pointeur_Int = ^integer;
    var ptr_entier : T_Pointeur_Int;
begin
    new(ptr_entier);
    ptr_entier^ := 10;
    writeln('La valeur est de ', ptr_entier^);
    dispose(ptr_entier);
end.
```

Pointeurs : déclaration et allocation (2)

`T_Pointeur_Int = ^integer ;` définit un nouveau type nommé `T_Pointeur_Int`, qui sera un *type pointeur* vers des entiers.

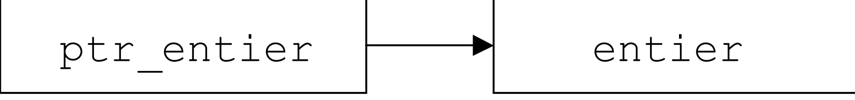
`ptr_entier : T_Pointeur_Int ;` déclare une variable de ce type. Cette déclaration *n'effectue pas de réservation mémoire* pour une valeur entière comme le ferait, par exemple une déclaration `entier : integer ;`. Elle réserve bien un emplacement nommé `ptr_entier` mais uniquement pour pouvoir y placer plus tard une adresse : celle de la valeur pointée.

• *à ce stade, la situation est donc la suivante :*

ptr_entier →

`new(ptr_entier) ;` alloue *dynamiquement* une zone de mémoire (zone réservée au cours de l'exécution du programme, et non à sa compilation). La variable `ptr_entier` pointe sur cette zone qui pourra être utilisée comme n'importe quel entier.

• *on a donc la situation suivante :*

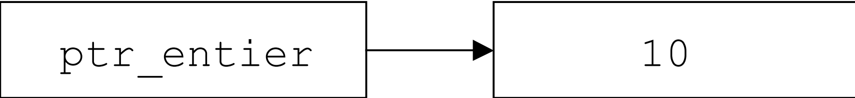


Pointeurs : déclaration et allocation (3)

<u>Rq</u> : Par rapport à nos explications sur le segment de données, le pointeur lui-même est alloué dans la partie statique du segment de données tandis-que l'emplacement pointé est alloué sur le tas.

`ptr_entier^ := 10 ;` signifie littéralement *" affecte 10 à l'emplacement mémoire pointé par* `ptr_entier` *"*.

- on a donc :



`dispose(ptr_entier) ;` est l'instruction duale de `new(ptr_entier) ;` : elle libère (désalloue) l'espace mémoire pointé par `ptr_entier`. Cela signifie qu'une référence à `ptr_entier` n'est plus valide... Nous sommes revenus à la situation de départ.

ptr_entier →

Pointeurs : utilisation sans allocation (1)

On peut également utiliser les pointeurs sans avoir recours à l'allocation dynamique : ils servent alors d'alias pour des variables déjà créées.

Pour ce faire, on peut utiliser l'opérateur d'indirection : placé devant un nom de variable, il renvoie l'adresse mémoire de celle-ci, qui peut donc être affectée à un pointeur. La forme générale est : `ptr := @variable ;`

Rq : Les pointeurs sont typés ce qui signifie qu'ils ne peuvent recevoir que des adresses d'emplacements du type pour lequel ils ont été définis. On ne peut affecter une adresse de réel à un pointeur d'entier, par exemple.

Exemple :

```
program POINTEURS_1;
    { Exemples simples sur les pointeurs }
var
            { déclaration de deux variables entières }
    entier_1, entier_2 : INTEGER;
            { déclaration de deux pointeurs vers des entiers }
    ptr_entier_1, ptr_entier_2 : ^INTEGER;
```

Pointeurs : utilisation sans allocation (2)

```
begin
    entier_1 := 10; entier_2 := 100;
    writeln('entier_1 = ', entier_1); writeln('entier_2 = ', entier_2);
        { les pointeurs reçoivent les adresses des variables }
    ptr_entier_1 := @entier_1; ptr_entier_2 := @entier_2;
        { on affiche les valeurs pointées par les pointeurs }
    writeln('ptr_entier_1 pointe sur ', ptr_entier_1^);
    writeln('ptr_entier_2 pointe sur ', ptr_entier_2^);
    ptr_entier_1 := ptr_entier_2;
        { ptr_entier_1 pointe maintenant sur le même emplacement que ptr_entier_2...}
    writeln('ptr_entier_1 pointe sur ', ptr_entier_1^);
    writeln('ptr_entier_2 pointe sur ', ptr_entier_2^);
    ptr_entier_1^ := 2000;
        { modifie l'emplacement pointé par ptr_entier_2 (donc entier_2)...}
    writeln('entier_1 = ', entier_1); writeln('entier_2 = ', entier_2);
        { Attention à ne pas confondre pointeurs et valeurs pointées ! }
    ptr_entier_1^ := ptr_entier_1^ + entier_1;
    writeln('entier_2 = ', entier_2);      ⇨
    entier_2 := ptr_entier_2^ * 10;
    writeln('entier_2 = ', entier_2);      ⇨
end.
```

quelles sont les 2 valeurs affichées pour entier_2 ?

Pointeurs : la valeur < `nil` > (1)

Lorsqu'un pointeur ne référence rien, il est préférable de l'indiquer explicitement en lui affectant la valeur prédéfinie `nil`, compatible avec n'importe quel type de pointeur.

Pour tester si un pointeur pointe vers un emplacement, il suffira alors de tester s'il est égal à `nil` :

```
if Ptr_Int = nil then ...
```

Un pointeur valant `nil` existe (il est déclaré) mais ne pointe sur rien.

Cela signifie notamment que si on affecte `nil` à un pointeur pour lequel on a déjà fait un new, l'emplacement qu'il désignait est perdu :

- on ne connaît plus son adresse (et donc la valeur) et,
- il n'y a plus moyen de faire un dispose pour le libérer.

Un `dispose(pointeur);` lorsque pointeur vaut `nil` est sans effet.

Pointeurs : la valeur < `nil` > (2)

Exemple :

```
program pointeurs2;

    type T_Pointeur_Entiers = ^integer;
    var Ptr_Int : T_Pointeur_Entiers;
begin
            {allocation de l'emplacement du pointeur}
    new(Ptr_Int);
            {affectation à 10}
    Ptr_Int^ := 10;
⇨   writeln('La valeur est ', Ptr_Int^);
            {libération de l'emplacement mémoire}
    dispose(Ptr_Int);
            {affectation à nil}
    Ptr_Int := nil;
    if Ptr_Int = nil then
        writeln('Le pointeur ne référence plus aucune variable')
    else
⇨       writeln('La valeur pointée par Ptr_Int est ', Ptr_Int^);

end.
```

quelles sont les 2 valeurs affichées pour `Ptr_Int^` ?

Pointeurs : comparaison et affectation

Un pointeur d'un type T peut être comparé avec tout autre pointeur de type T ou avec la valeur `nil`. Il en va de même pour l'affectation.

Exemple :

```
program pointeurs2;

    type    T_Pointeur_Entiers = ^integer;
    var     Ptr_Int1, Ptr_Int2 : T_Pointeur_Entiers;
begin
        {allocation de Ptr_Int1}
    new(Ptr_Int1);
        {affectations des deux pointeurs}
    Ptr_Int1^ := 10;
    Ptr_Int2 := Ptr_Int1;
        {affichage de Prt_Int2}
⇨   writeln('Ptr_Int2 pointe sur ', Ptr_Int2^);
    dispose(Ptr_Int1);
⇨   writeln('Ptr_Int2 pointe sur ', Ptr_Int2^);

end.
```

quelles sont les 2 valeurs affichées pour `Ptr_Int2^` ?

Pointeurs : résumé des notions de base

CONSTANTE

Nil La valeur `nil` est compatible avec tous les types de pointeurs. Elle sert à faire pointer un pointeur sur rien. `nil` peut être utilisée dans les tests sur les variables pointeurs.

OPERATEUR @

L'opérateur @ est un opérateur unaire qui est couplé soit à une variable, soit a un identificateur de procédure ou de fonction. Le résultat est un pointeur sur l'opérande.
Le pointeur est de même type que `nil`, c'est pourquoi il peut être affecté à toute variable de type pointeur.

FONCTIONS

Addr Equivalent à @.

PROCEDURES

Dispose(ptr); libère la mémoire correspondant au pointeur `ptr`.
New(ptr); Crée une variable associée au pointeur de variable `ptr`.
La variable créée sera référencée par `ptr^`.

Les structures récursives

Une ***structure récursive*** est une structure dont l'un des champs est du type de la structure.
La définition suivante n'est cependant pas correcte :

```
type    T_Personne = record
                nom : string;
                pere, mere : T_Personne;
        end;
```

Rq : `T_Personne` n'est connu qu'après le `end;` de la définition : Pascal ne connaît donc pas la taille mémoire qu'il faut réserver pour les champs `pere` et `mere`.

Les pointeurs, et la possibilité d'anticiper la définition d'un type permettent de corriger ce problème :

```
type    T_Ptr_Personne = ^T_Personne;
        T_Personne = record
                nom : string;
                pere, mere : T_Ptr_Personne; {pointeur}
        end;
```

C'est cette méthode que l'on utilisera pour toutes les structure de données gérer dynamiquement, telles que les ***listes***.

La liste simplement chaînée (1)

Une liste simplement chaînée est constituée d'enregistrements (cellules) liés entre eux par des pointeurs. Vous pouvez penser à un train comme à une liste, les wagons étant les cellules liées les unes aux autres et contenant les informations (type de wagon, liste des passagers, …).

Chaque enregistrement contiendra deux champs :

1. ***la donnée utile***, et
2. ***un lien*** qui pointera vers l'enregistrement suivant.

Une cellule permettant de stocker des entiers peut donc être définie ainsi :

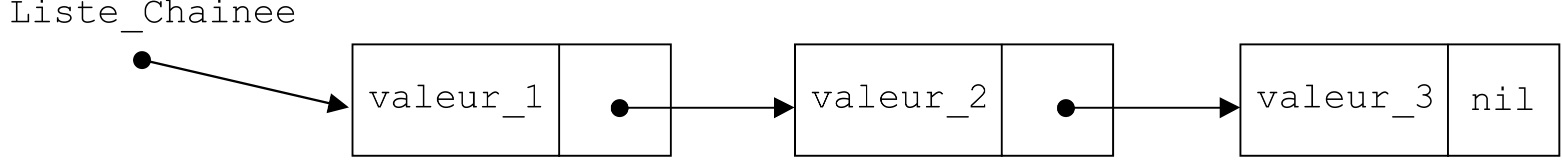
```
type    T_Ptr_Cellule = ^T_Cellule;
        T_Cellule = record
                valeur : integer;
                suivant : T_Ptr_Cellule; {pointeur}
        end;
```

Pour déclarer une telle liste, on définira une variable `Liste_Chainee`, de type `T_Ptr_Cellule`.

```
var      Liste_Chainee : T_Ptr_Cellule;
```

La liste simplement chaînée (2)

Une liste de trois éléments peut donc se représenter visuellement de la façon suivante :



Vous noterez que le dernier champ `suivant` vaut `nil`, ce qui signifie qu'il s'agit de la dernière cellule.
Selon le même principe, une liste vide sera repérée par le fait que son pointeur de tête (`Liste_Chainee`, ici) vaudra `nil`.

Rq : il ne faudra donc pas oublier d'initialiser ce dernier pointeur à `nil` (soit le pointeur de tête pour une liste vide, soit le dernier champ `suivant`).

Liste chaînée : insertion (1)

Dans tous les cas l'insertion se fait en 2 temps :

1. ***on crée une nouvelle cellule*** avec l'instruction `new(Nouveau);` et on initialise le champ valeur de cette cellule avec la valeur voulue.
2. ***on la place dans la liste***, pour cela il faut distinguer plusieurs cas.

a) Première insertion

Si la liste est vide, le pointeur qui la désigne contient la valeur nil : il suffit donc de lui affecter `Nouveau`.

```
Liste_Chainee := Nouveau;
```

b) Insertion en tête de la liste

Il s'agit ici de modifier le pointeur de tête de liste et d'établir un lien entre la nouvelle cellule et celle qui se trouvait en tête :

```
{ On établit le lien }
    Nouveau^.Suivant := Liste_Chainee;
{ On modifie le pointeur de tête }
    Liste_Chainee := Nouveau;
```

Liste chaînée : insertion (2)

c) Insertion au milieu de la liste

Dans une liste simplement chaînée, le sens de parcours est toujours d'une cellule vers sa suivante : *on ne peut revenir en arrière*.

Pour insérer entre deux cellules, on insèrera donc toujours après une cellule donnée.

Soit `Courant` la cellule après laquelle on insère :

- on mémorise ***d'abord*** la cellule suivante de `Courant` dans le champ `Suivant` de `Nouveau` :

```
Nouveau^.Suivant := Courant^.Suivant;
```

!

- on peut donc maintenant faire pointer le champ `Suivant` de `Courant` sur `Nouveau` : `Courant^.Suivant := Nouveau;`

d) Insertion en fin de liste

Il s'agit d'un cas particulier du précédent : insérer en fin de liste, c'est insérer après la dernière cellule. Dans ce cas, le champ `Suivant` de `Courant` vaut `nil`.

Après insertion, le champ `Suivant` de `Nouveau` vaudra `nil` et celui de `Courant` pointera sur `Nouveau`.

Liste chaînée : suppression (1)

Dans tous les cas, on suppose que `Temp` pointe la cellule à supprimer.

Le sens de parcours étant unidirectionnel : on supprimera donc toujours `Temp` en utilisant la cellule qui la précède (`Courant`), sauf s'il s'agit de la première cellule de la liste.

*Rq : Comme pour l'insertion, on traitera à part la suppression de la première cellule car c'est **le seul cas** où on modifie directement `Liste_Chainee`.*

a) Suppression en tête de liste

On libère le premier emplacement après avoir mémorisé le suivant :

```
Temp := Liste_Chainee^.suivant;
dispose(Liste_Chainee);
Liste_Chainee := Temp;
```

Rq : On notera que ce cas traite la situation où `Liste_Chainee` est constituée d'une seule cellule, `Temp` vaudrait alors `nil`.
Après la suppression, `Liste_Chainee` vaudrait également `nil` : la liste serait considérée comme vide.

Liste chaînée : suppression (2)

b) Suppression au milieu de la liste

- on mémorise l'emplacement de la cellule à supprimer :

```
Temp := Courant^.Suivant;
```

- on établit le lien entre Courant et la cellule suivant celle que l'on va supprimer :

```
Courant^.Suivant := Temp^.Suivant;
{ou Courant^.Suivant^.Suivant}
```

- on libère l'espace alloué à la cellule à supprimer :

```
dispose(Temp); {impossible si on a pas mémorisé Temp}
```

c) Suppression en fin de liste

Ici, `Temp.Suivant = nil` : après suppression, `Courant.Suivant` vaudra donc `nil`, ce qui indiquera que `Courant` sera la dernière cellule de la liste.

d) Suppression d'une liste chaînée entière

La suppression d'une liste complète passe par la suppression de tous ses éléments, ***dans le bon ordre***. À l'issue de cette opération, la liste doit être vide ***et*** l'ensemble de l'espace mémoire qu'elle occupait libéré.

(L'écriture de cette opération est laissée à titre d'exercice.)

Approche d’autres structures récursives

Nous avons abordé la liste simplement chaînée qui est la structure récursive la plus simple, mais bien d’autres existent dont certaines sont très utilisées, en voilà quelques exemples:

- *liste doublement chaînée*
- *pile (LIFO)*
- *file (FIFO)*
- *anneau*
- *arbre de calcul*
- *…*

Nous aborderons la ***pile*** en TP, c’est une structure très classique à la base de certaines méthodes de calcul comme la notation polonaise.

L’ensemble de ces structures de données restent basées sur la notion de pointeur faisant le lien entre différents enregistrements.

Conclusion

Les pointeurs sont un outil ***essentiel*** à tout programmeur, ils permettent la gestion dynamique de la mémoire et des structures de données nécessaires à bien des programmes.

Il est essentiel de bien avoir compris leur fonctionnement et les principe des structures de données récursives qui sont elles-mêmes basées sur les pointeurs.

Vous êtes donc encouragés à bien relire ce cours et les exemples présentés, ces aspects seront repris et approfondis en TD.