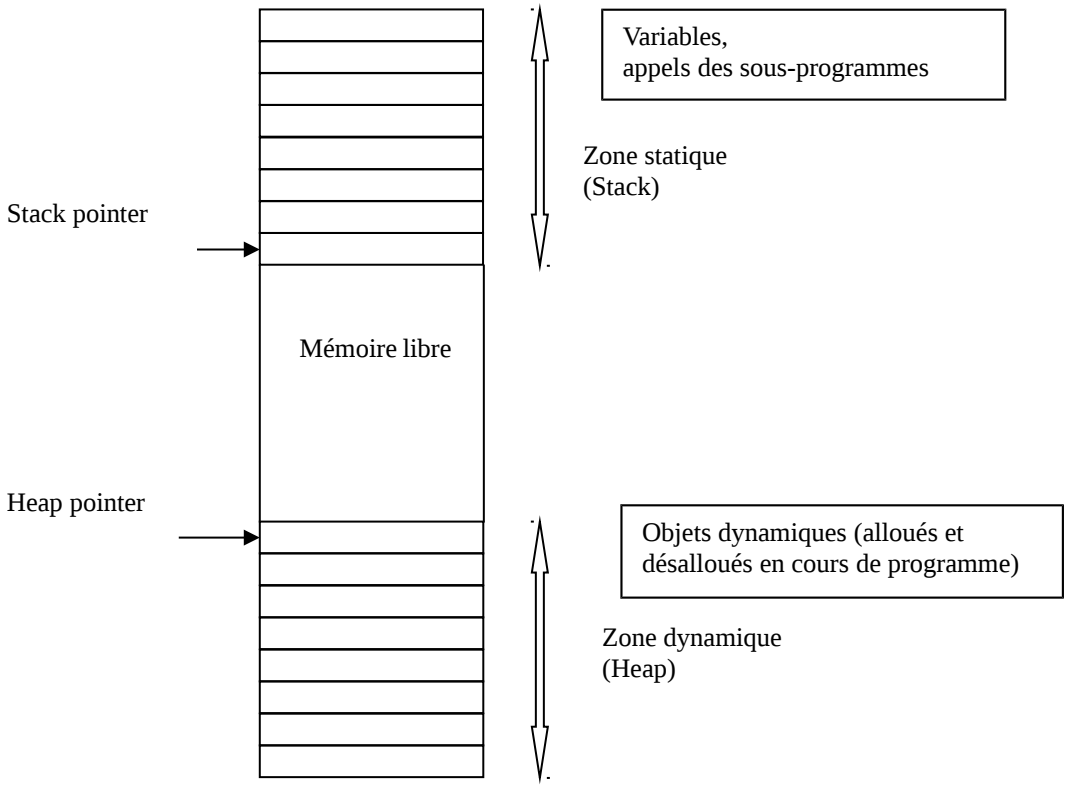
Algorithmique & Structures de données

Pointeurs

Dans ce chapitre, nous examinerons comment réaliser des piles dont la taille varie au cours de l'exécution d'un programme. Nous ferons usage d'une particularité que possède presque tous les langages : pouvoir gérer une zone de mémoire allouée dynamiquement en cours d'exécution et à la demande du programme.

Schématiquement, on peut représenter cela de la manière suivante :



Le système d'exploitation gère la zone dynamique : dès qu'une demande d'allocation est faite par l'intermédiaire de fonctions appropriées comme **`malloc`** en C, **`new`** en Ada, la quantité de mémoire nécessaire est attribuée et le pointeur « heap » est remis à jour.

Si la zone mémoire n'est plus utilisée, il est possible de la désallouer et de la rendre au système. Ceci est de la responsabilité du programmeur en Ada, contrairement à Java où la restitution de la mémoire se fait automatiquement par l'intermédiaire d'un « ramasse-miette » (*garbage collector*).

En Ada, un type spécial de variables est fourni pour référencer des données stockées dans la zone dynamique : ce sont les variables de type **`access`**. On parle aussi de variables dynamiques (cela correspond au type pointeur des autres langages). Les opérations possibles sur les pointeurs sont :

- l'affectation
- le passage en paramètres
- l'allocation avec **`new`**
- la libération de la mémoire
- l'accès au contenu de l'emplacement pointé avec **`all`**

Syntaxiquement, la manipulation des variables de type pointeur peut varier d'un langage à l'autre mais l'idée est la même : allouer la mémoire uniquement en cas de besoin et libérer la place non utilisée.

Exemple

On définit le type tableau suivant :

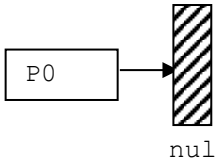
```
type T_Tab is array(1..10) of Integer;
```

Voici quelques déclarations de types pointeurs :

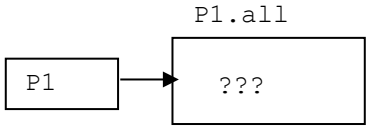
```
type T_Pt_Int is access Integer;
type T_Pt_Flt is access Float;
type T_Pt_Tab is access T_Tab;
```

des déclarations et initialisations de variables de type pointeur:

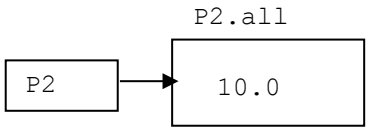
```
P0 : T_Pt_Int;
```



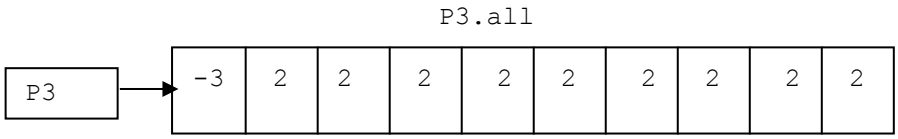

```
P1 : T_Pt_Int := new Integer;
```




```
P2 : T_Pt_Flt := new Float'(10.0);
```

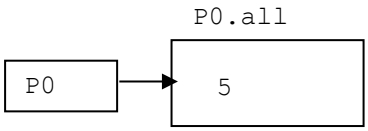



```
P3 : T_Pt_Tab := new T_Tab'((1 => -3, others => 2));
```

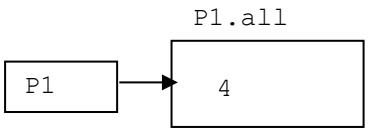


et des instructions:

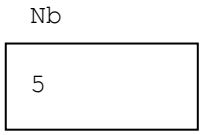
```
P0        := new Integer'(5);
```



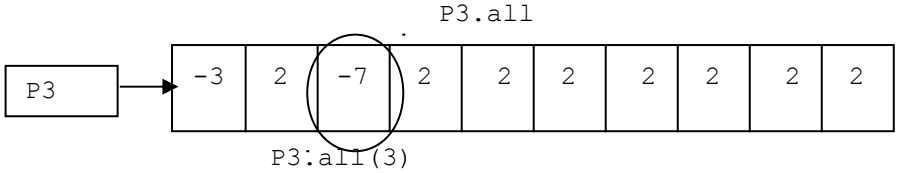

```
P1.all    := 4;
```




```
Nb        := P0.all;   -- on suppose qu'on a déclaré Nb : Integer;
```




```
P3.all(3) := -7;
```




```
Put(P2.all); -- 10.0 est affiché à l'écran
```

Exemple

Définissons un type article propre à recevoir une donnée entière ainsi qu'un caractère

```
type RecType is record
     Valeur         : Integer;
     Identificateur : Char;
end record;
```

Un type de variable dynamique permettant de référencer une telle donnée sera déclarée par

```
type RecPointer is access RecType;
```

A partir de ce moment, nous pouvons déclarer des variables de type RecPointer c'est-à-dire des variables capables d'accéder à des données de type `RecType`.

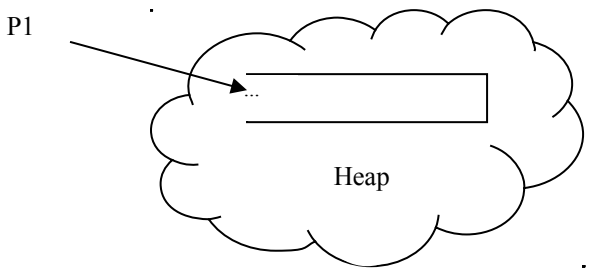
```
P1, P2, P3 : RecPointer;
```

Une fois ces variables déclarées, leur valeur initiale est la constante **`null`** ; cela signifie que ces variables ne référencent rien actuellement (pas d'allocation).

L'allocation se réalise à l'aide de l'opérateur **`new`**, une instruction telle que :

```
P1 := new RecType;
```

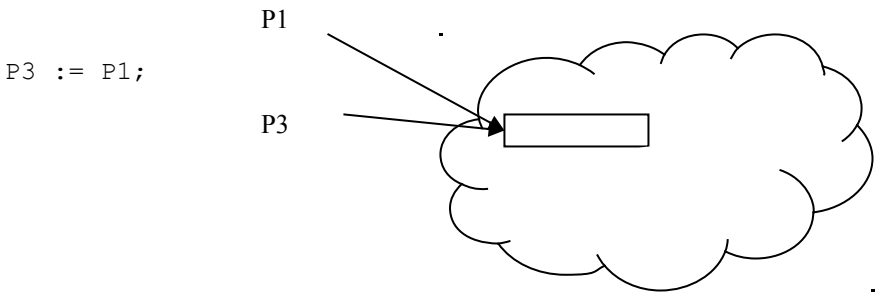
alloue un bloc de mémoire de taille suffisante pour contenir les données de ce type, soit un entier et un caractère dans ce cas particulier. A noter que le contenu du bloc mémoire n'a pas encore été initialisé.

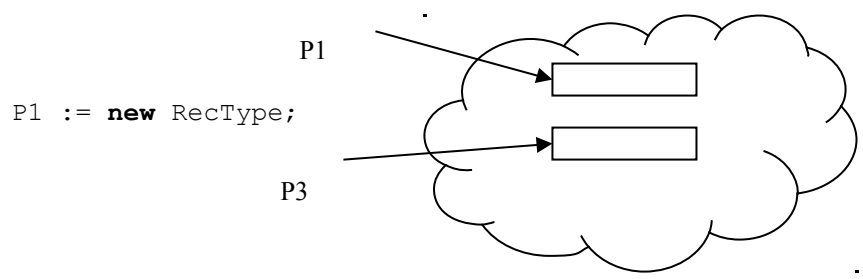


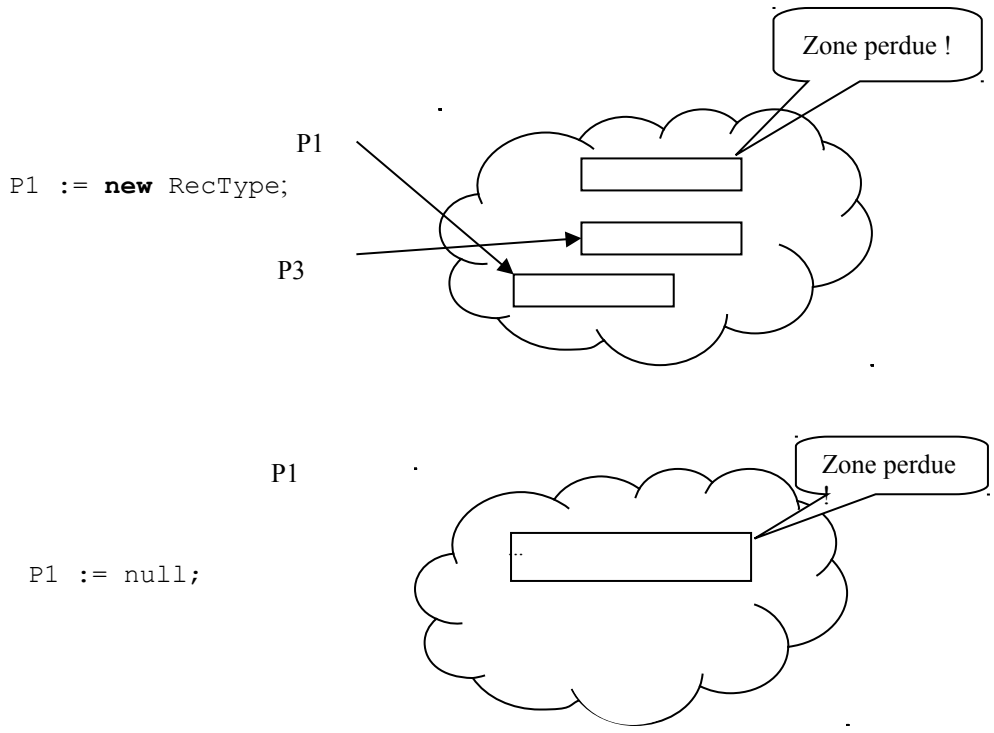
A partir de là, on peut utiliser cette zone de mémoire pour y stocker de l'information.

On peut effectuer des opérations d'affectation et des comparaisons entre variables de type **`access`**. Les seules affectations possibles le sont entre variables qui référencent un même type ou l'affectation avec la valeur **`null`**.

La figure suivante illustre le fonctionnement de l'allocation pendant l'exécution (*run time allocation*).







L'accès aux données

Une fois la place allouée, l'accès aux données se fait via le nom de la variable de type accès comme par exemple dans la séquence suivante :

```
P1 := new RecType ;      -- allocation dynamique
P1.all.valeur := 123;    -- accède au champ valeur de la donnée
                         -- pointée P1.all
P1.all.identificateur := 's';  -- idem pour le champ identificateur
```

Dans le cas où la donnée pointée est un article, on peut utiliser une forme simplifiée et écrire :

```
P1.valeur := 123;
P1.identificateur := 's';
```

On peut regrouper allocation et affectation en une seule instruction :

```
P1 := new RecType'((valeur => 123, identificateur => 's'));
```

On peut naturellement lire ou afficher les données :

```
Integer_Text_IO.Put(P1.valeur);
Text_IO.Put(P1.identificateur);
```

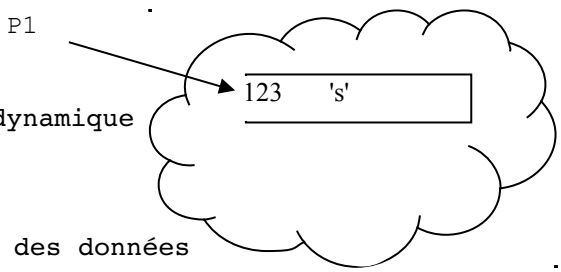
Au moment où les données ne sont plus utiles, on peut libérer la mémoire en utilisant une procédure de libération des variables dynamiques d'un type donné.

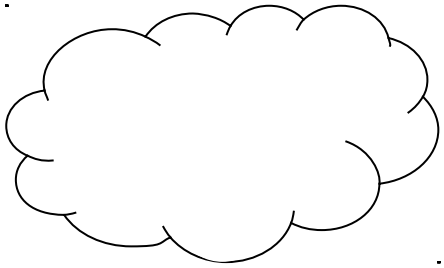
Pour cela, on se sert d'un modèle (procédure générique prédéfinie) que l'on instancie en l'adaptant aux types de donnée que l'on doit traiter (il faut que la taille de la zone à libérer soit connue).

Cette procédure se nomme **`Unchecked_Deallocation`** et pour libérer la zone référencée plus haut par `P1` on écrira la séquence suivante:

```
with Unchecked_Deallocation; -- dans la partie de clause de contexte
...
procedure Liberer is new Unchecked_Deallocation(RecType,RecPointer);
...
```

```
-- la séquence d'allocation

P1 := new RecType ;     -- allocation dynamique
P1.all.valeur := 123;
P1.all.identificateur := 's';

-- le reste du programme d'utilisation des données
...

--l'instruction de libération

Liberer(P1);
```

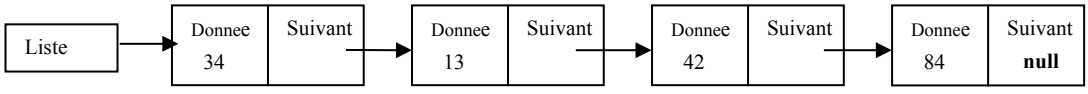




La zone occupée précédemment par les données est rendue au système et `P1` prend la valeur **`null`.** Il faut noter que, comme `P1` n'est plus alloué, le bloc mémoire `P1.all` n'est pas accessible et une instruction contenant `P1.all` provoquera une erreur à l'exécution (et non à la compilation). Si `P2` est de type accès sur un entier, alors l'affection `P1 := P2` provoque une erreur à la compilation, car on ne peut affecter que des pointeurs de même type.

Exemple : la déclaration d'une pile dynamique

Une pile sera réalisée à l'aide d'une liste chaînée d'articles. Chaque article comportera un champ pour stocker les valeurs dans la pile et un champ contenant une variable de type accès de façon à assurer le chaînage.
Schématiquement:

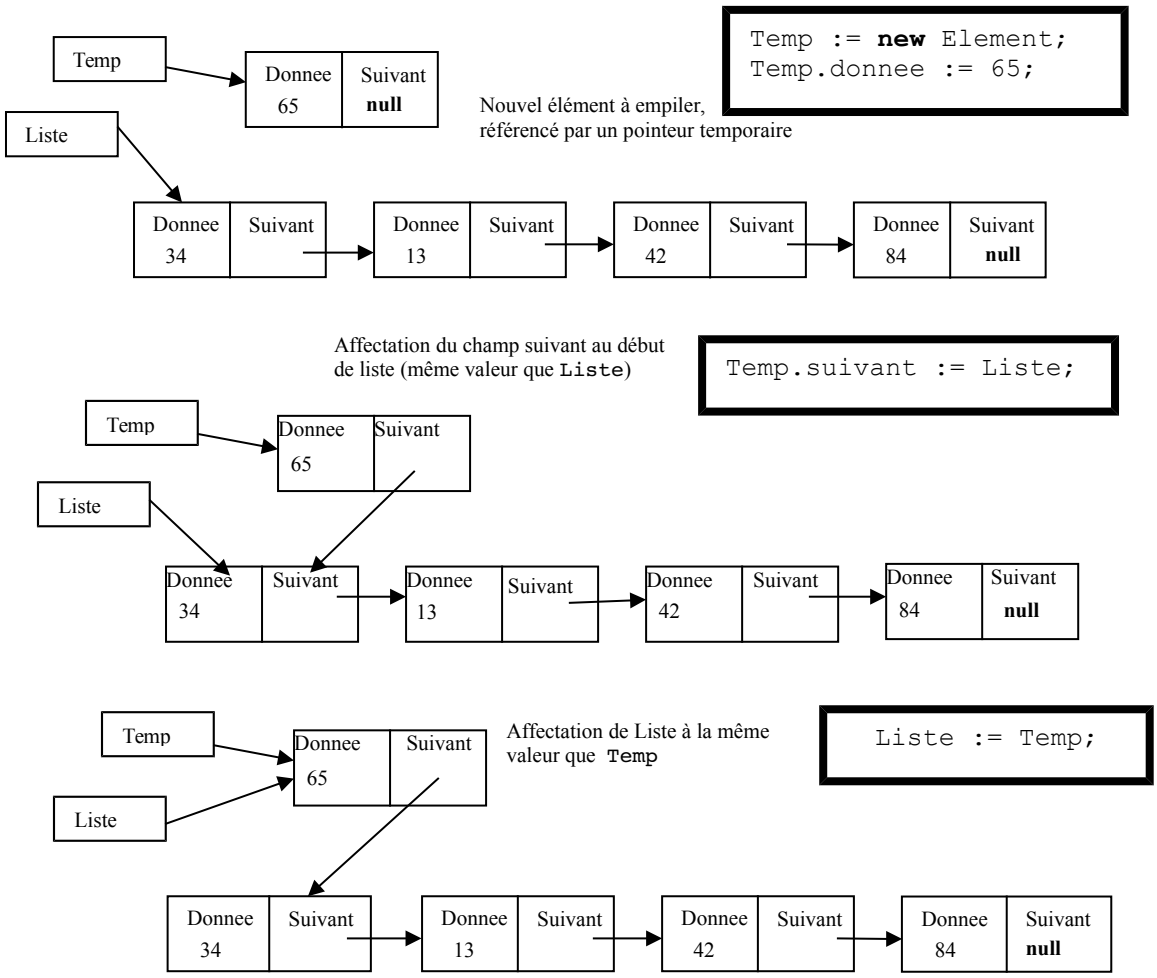


L'accès à la pile se fera uniquement par le pointeur `Liste` qui référence le début de la liste et en même temps sommet de la pile.

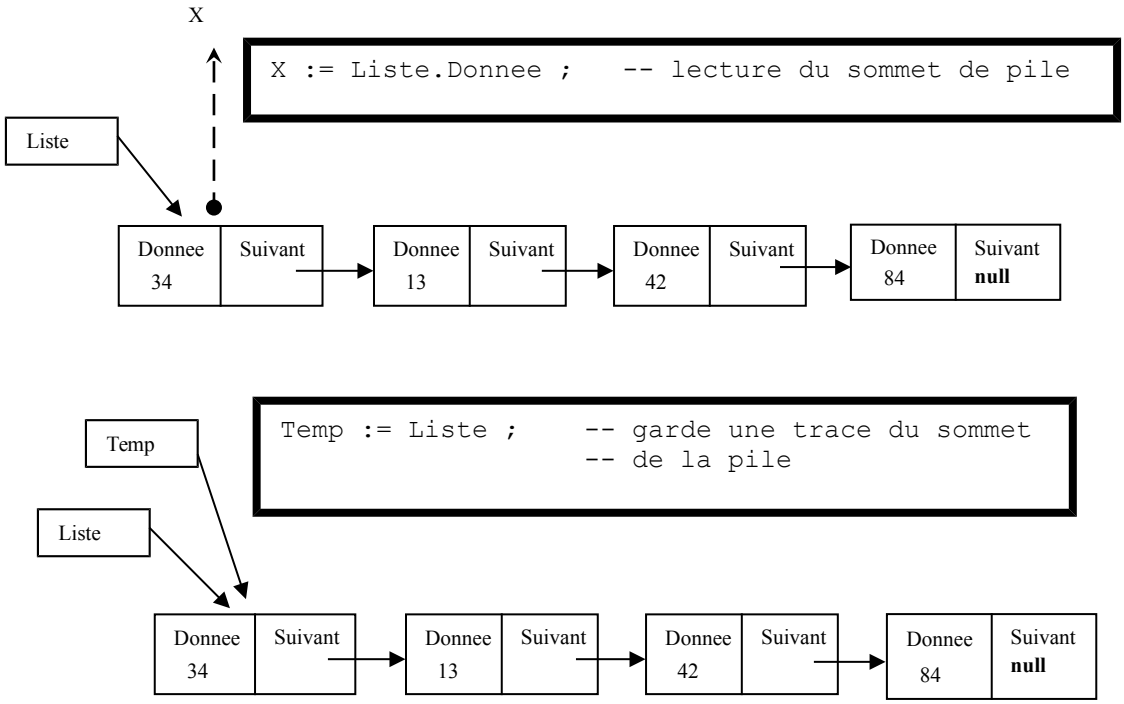
Une telle liste chaînée liste sera déclarée par exemple sous la forme suivante:

```
type Element;                              -- pré-déclaration indispensable !
type Pointeur_Element is access Element; -- pointeur des éléments de liste
type Element  is record                    -- élément de liste
     Donnee  : Integer ;
     Suivant : Pointeur_Element := null;
end record;
```

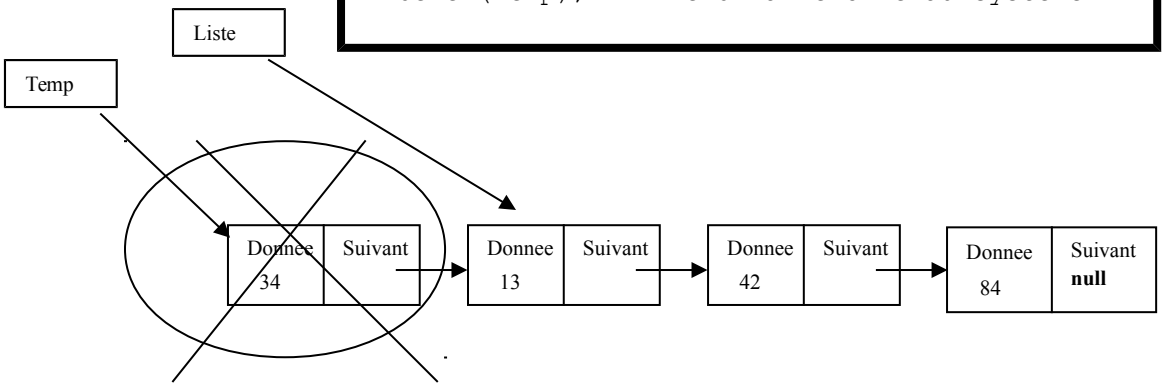
L'empilement d'un élément se fera en tête de liste



Pour retirer un élément du sommet de la pile, on procède de manière analogue en prenant garde toutefois de libérer la place occupée par l'élément que l'on a retiré.



```
Liste := Liste.suivant;  -- pointe sur l'élément
                         -- suivant dans la pile
Liberer(Temp);  -- rend la mémoire au système
```



Exercice 1

Donnez la valeur pointée par la variable (`P1`, `P2`, `P3`, `P4`, `P5` et `P6`) après chaque instruction, aidez-vous de dessins.

```
procedure Pointeurs is
   type T_Pt_Character is access Character;
   P1, P2, P3, P4, P5, P6 : T_Pt_Character := null;
begin
   P1 := new Character'('N');
   P1.all := Character'Val(Character'Pos(P1.all)+1);
   P2 := new Character;
   P2.all := 'M';
   P3 := new Character'('U');
   P4 := P3;
   P4.all := 'I';
   P4 := P5;
   P5 := P1;
   P6.all := 'R';
end Pointeurs;
```

Exercice 2

Soit le type :

```
type T_Pt_Int is access Integer;
```

Ecrire une procédure qui échange les valeurs pointées par deux pointeurs :

```
procedure Echanger (P1,P2 : in out T_Pt_Int);
```

Ecrire deux fonctions qui effectuent une addition :

```
function “+” (P1,P2 : T_Pt_Int) return Integer;
function “+” (P1,P2 : T_Pt_Int) return T_Pt_Int;
```

Définir les variables nécessaires et écrire l’instruction qui permettent d’utiliser chacune des deux fonctions.

Exercice 3

Ecrire un paquetage complet de gestion de pile dynamique.