Analyse et programmation langage ADA

Informatique 2ème année

type access

- **allocation dynamique de mémoire**
- **restitution, libération, ramasse-miettes**
- **pointeurs**

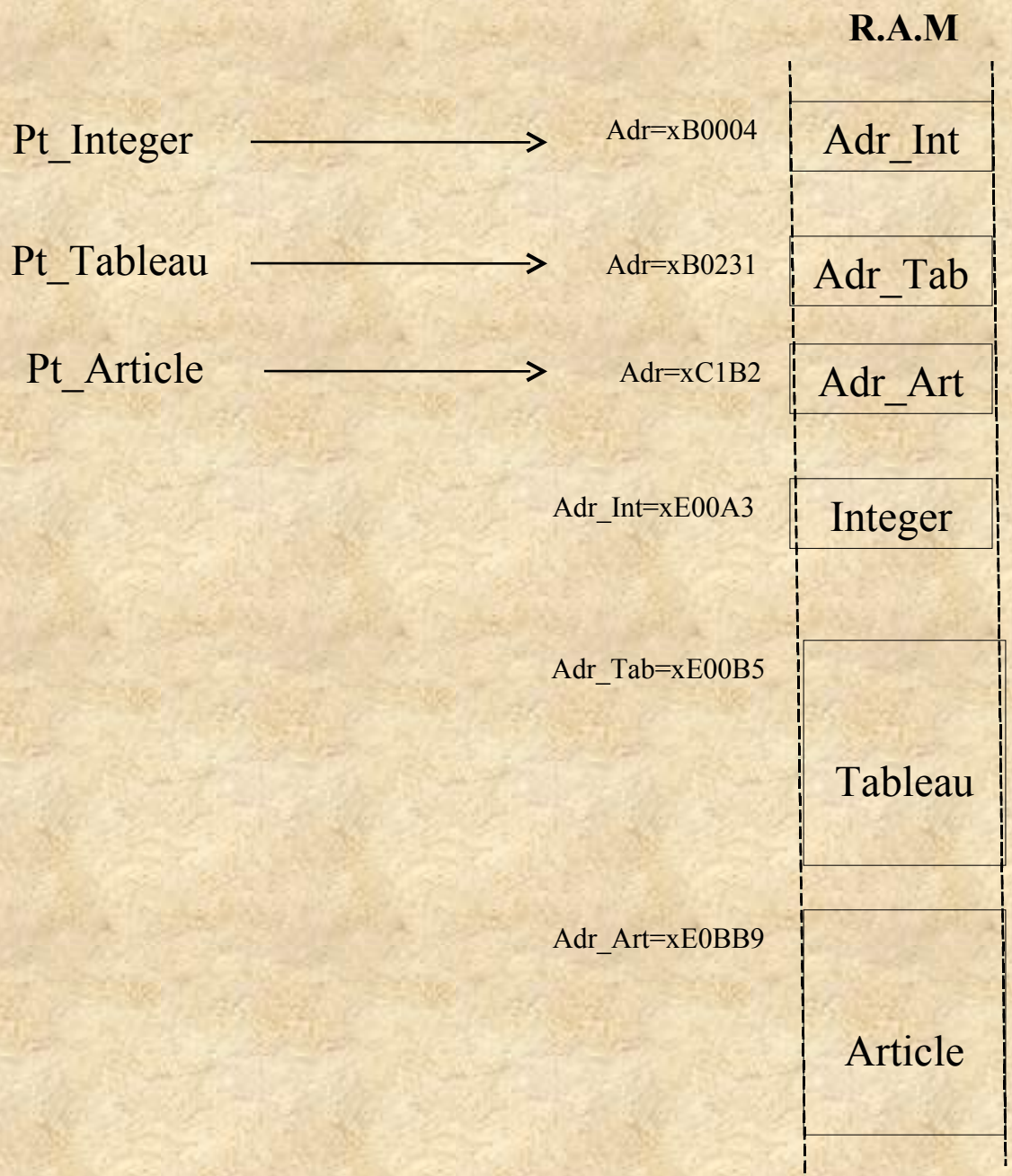
type identificateur_type_acces **is access** identificateur_de_type_ou_sous_type;

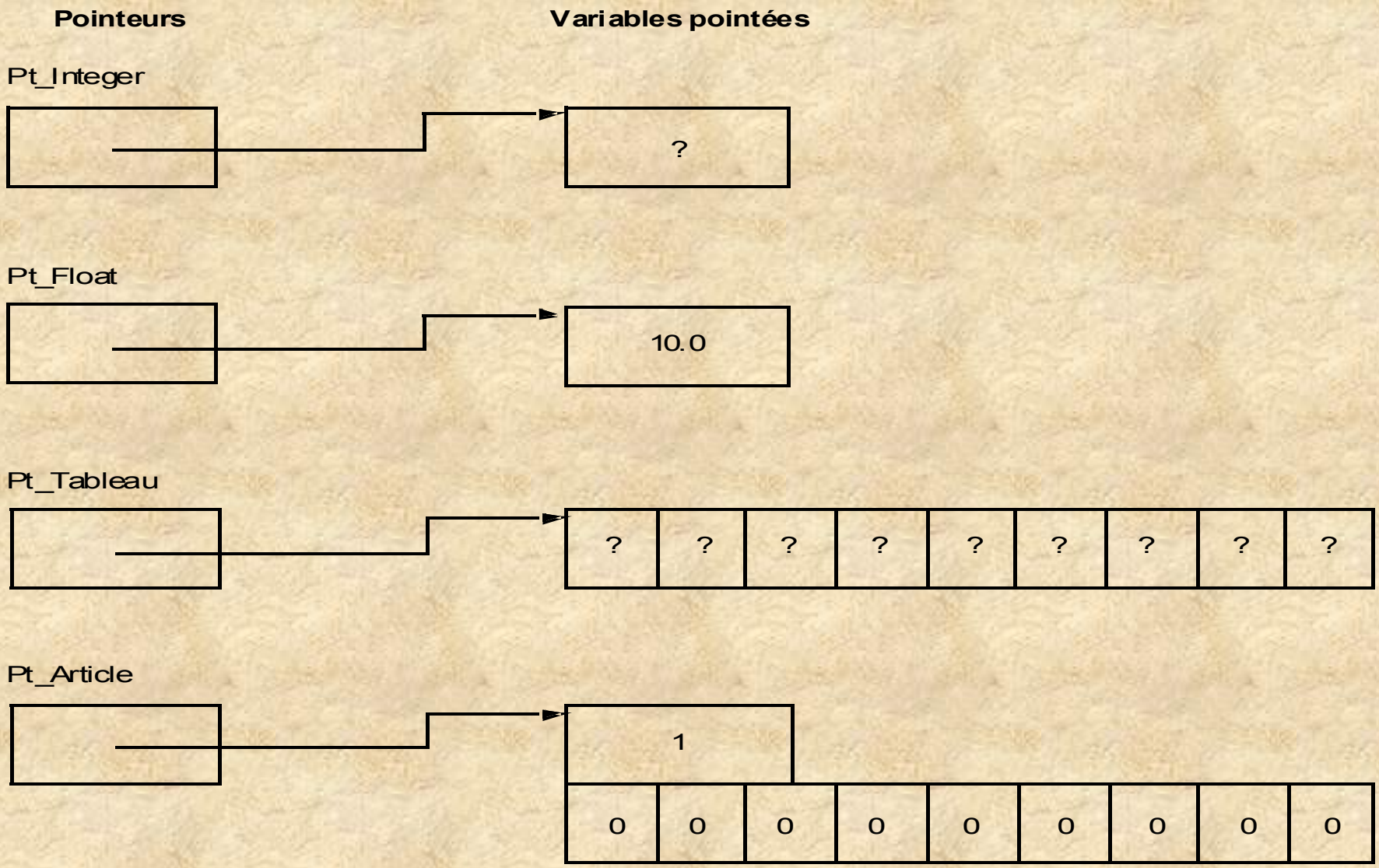
```
type T_Tableau is array (1..9) of Integer;          -- Un type tableau
type T_Article is                                   -- Un type article
    record
        Nombre : Integer;
        Tab : T_Tableau;
    end record;

type T_Pt_Integer is access Integer;        -- Un type pointeur sur Integer
type T_Pt_Float is access Float;            -- Un type pointeur sur Float
type T_Pt_Tableau is access T_Tableau;      -- Un type pointeur sur le type
                                            -- pointe T_Tableau
type T_Pt_Article is access T_Article;      -- Un type pointeur sur le type
                                            -- pointe T_Article
```

Allocateur new et affectation

new identificateur_de_type_ou_sous_type
new expression_qualifiee

```
Pt_Integer : T_Pt_Integer;          -- Une variable pointeur sur Integer
Pt_Float : T_Pt_Float;              -- Une variable pointeur sur Float
Pt_Tableau : T_Pt_Tableau;          -- Une variable pointeur sur T_Tableau
Pt_Article : T_Pt_Article;          -- Une variable pointeur sur T_Article

-- Utilisation de l'allocateur
Pt_Integer := new Integer;          -- Pt_Integer repere la variable
                                    -- pointee creee
Pt_Float := new Float'(10.0);       -- Pt_Float repere la variable pointee
                                    -- creee de valeur initiale 10.0
Pt_Tableau := new T_Tableau;        -- Pt_Tableau repere la variable
                                    -- pointee creee
Pt_Article := new T_Article'(1,(others=>0));  -- Pt_Article repere la
                                              -- variable pointee creee de
                                              -- valeur initiale (1, (others=>0) )
```

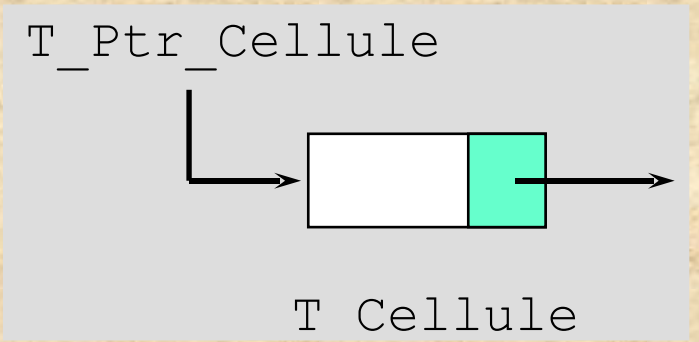

R.A.M
Pt_Integer
Adr=xB0004
Adr_Int
Pt_Tableau
Adr=xB0231
Adr_Tab
Pt_Article
Adr=xC1B2
Adr_Art
Adr_Int=xE00A3
Integer
Adr_Tab=xE00B5
Tableau
Adr_Art=xE0BB9
Article


Pointeurs
Variables pointées
Pt_Integer
?
Pt_Float
10.0
Pt_Tableau
? ? ? ? ? ? ? ? ?
Pt_Article
1
0 0 0 0 0 0 0 0 0

Utilisation classique des types accès

Pointeurs sur les objets

```
type T_Cellule;
type T_Ptr_Cellule is access T_Cellule;
```




```
type T_Cellule is
record
   Contenu      : Integer;  -- un ou plusieurs champs pour
                            -- l'information geree  dans la structure dynamique
   Ptr_Suivant  : T_Ptr_Cellule;  -- un ou plusieurs liens pour la
                                  --  realisation de la  structure dynamique
end record;

La_Tete, La_queue : T_Ptr_Cellule;
```

Utilisation classique des types accès

Nous pouvons définir une constante de type **access**:

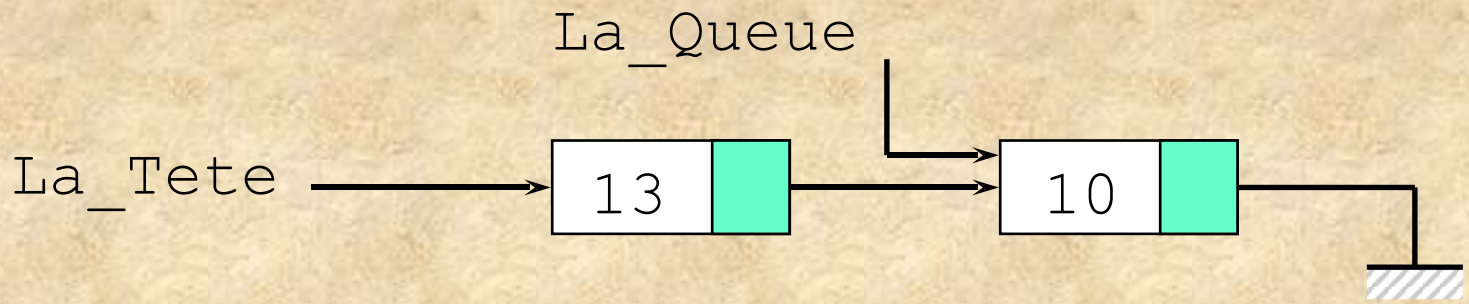
```
Tete_Vide : constant T_Ptr_Cellule := new T_Cellule'( 0, null );
```

Le pointeur T_Ptr_Cellule désignera toujours la même cellule, donc ne pourra pas changer de valeur, mais nous avons tout à fait le droit d'écrire:

```
Tete_Vide.Contenu := 33;
Tete_Vide. Ptr_Suivant := Tete_Vide;
Tete_Vide.all := La_Tete.all;
```

Allocation

```
La_Tete := new T_Cellule'(13, La_Queue);
La_Queue := new T_Cellule'(10, null);
```



Accès au contenu

```
La_Tete.Contenu                    -- c'est 13
La_Tete.Ptr_Suivant                -- c'est La_Queue
```

Copie les pointeurs et les contenus

```
La_Tete := La_Queue;               -- les pointeurs
La_Tete.all := La_Queue.all;       -- les contenus
```

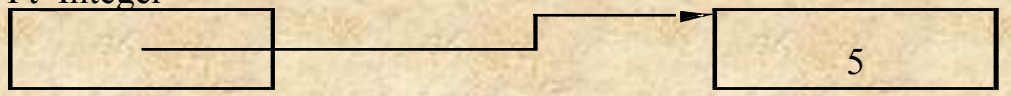
```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
procedure Exemple is
      Pt_Integer : T_Pt_Integer := new Integer;
      Pt_Float : T_Pt_Float := new Float'(10.0);
      Pt_Tableau : T_Pt_Tableau := new T_Tableau;
      Pt_Article : T_Pt_Article := new T_Article'(1, (others=>0));

      Entier : Integer;                    -- Deux variables auxiliaires
      Article : T_Article;

begin -- Exemple
      Pt_Integer.all := 5;                                          -- 1
      Entier := 3 * Pt_Integer.all - 1;                             -- 2
      Put ( Pt_Float.all );                                         -- 3
      Pt_Tableau.all := (1..4 =>0, 5|7|9 => 1, others => 2);    -- 4
      Article := Pt_Article.all;                                    -- 5
      Pt_Tableau (1) := 5;                                          -- 6 ,
      Pt_Tableau.all (1) := 5;                                      -- Identique a l'instruction 6
      for I in Pt_Tableau.all'Range loop                            -- 7
            Put ( Pt_Tableau (I) );
end loop;
```

```
Pt_Article.Nombre := 10;                -- 8
Pt_Article.all.Nombre := 10;            -- Identique a l'instruction 8
Pt_Article.Tab := Pt_Tableau.all;       -- 9
for I in Pt_Article.Tab'Range loop      -- 10
        Put ( Pt_Article.Tab (I) );
end loop;
...
```

Pointeurs **Variables pointées**

Instruction 1:

Pt_Integer

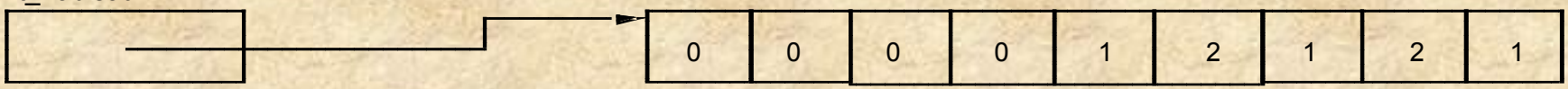


Instruction 2: la variable Entier reçoit la valeur 14

Instruction 3: affichage de la valeur 10.0

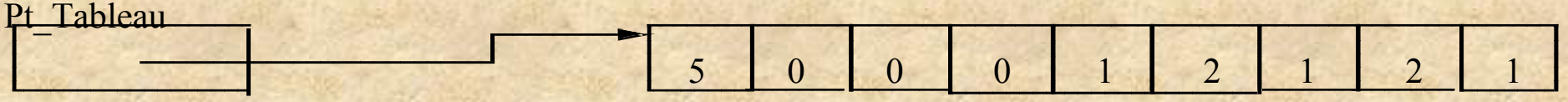
Instruction 4:

Pt_Tableau



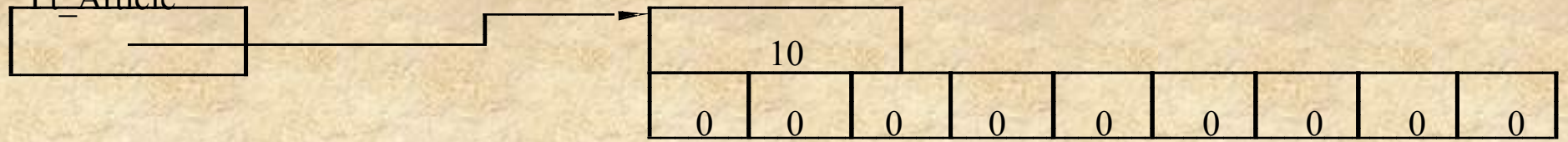
Instruction 5: la variable Article reçoit la valeur (1, (others => 0))

Instruction 6:

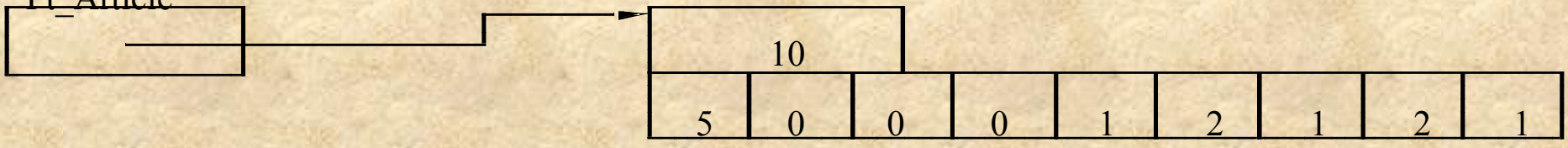


Instruction 7: affichage successif des valeurs 5 0 0 0 1 2 1 2 1

Instruction 8:



Instruction 9:



Instruction 10: affichage successif des valeurs 5 0 0 0 1 2 1 2 1

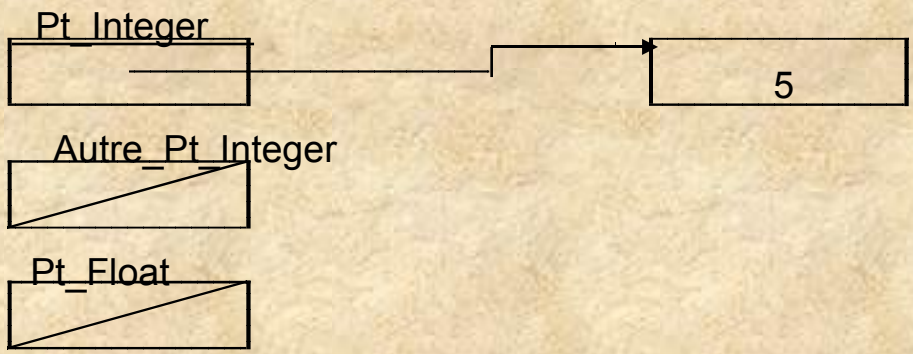
Affectation

```
Pt_Integer : T_Pt_Integer := new Integer'(5);        -- Trois variables pointeurs
Autre_Pt_Integer : T_Pt_Integer;
Pt_Float : T_Pt_Float;                               -- 1, situation initiale

Autre_Pt_Integer := Pt_Integer;                      -- 2
Pt_Integer.all := 1;                                 -- 3
-- que vaut Autre_Pt_Integer.all?                    -- 4
Autre_Pt_Integer := new Integer;                     -- 5
Autre_Pt_Integer.all := Pt_Integer.all;              -- 6
Pt_Float := Pt_Integer;                              -- 7, erreur de type, leve Constraint_Error
Pt_Float.all := 0.0;                                 -- 8,
```

Pointeurs

Variables pointées

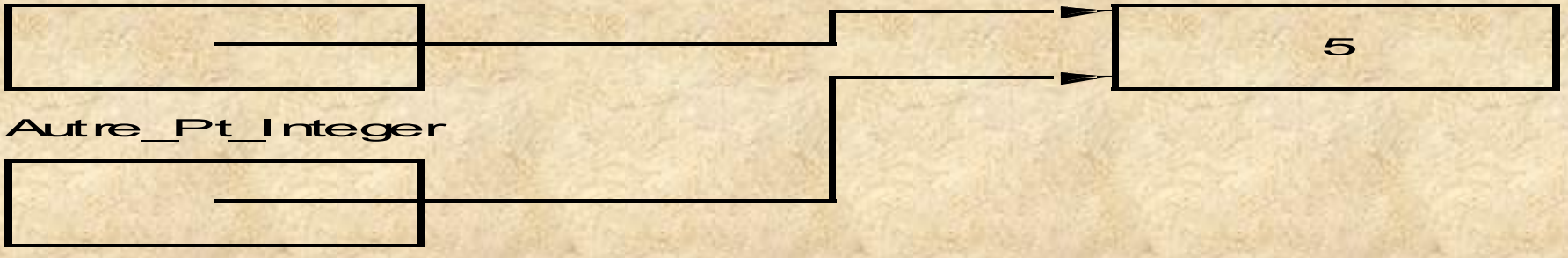
Ligne 1, les 3 déclarations:



Instruction 2 :

Pt_Integer

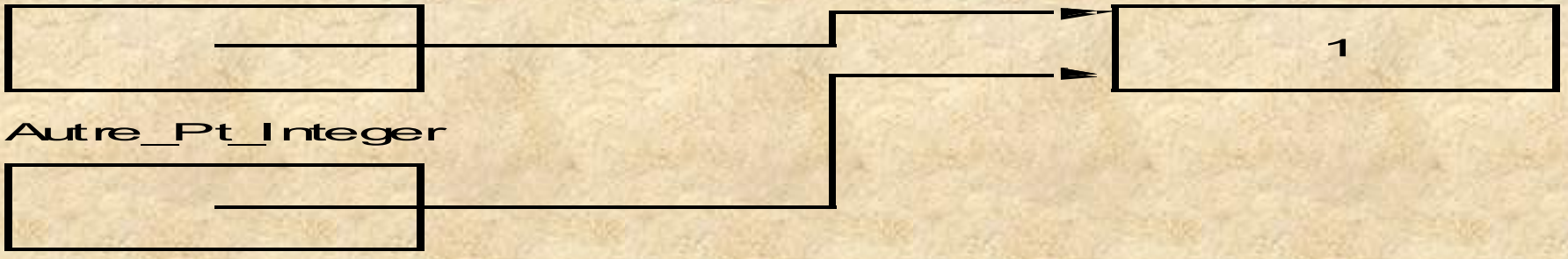
Autre_Pt_Integer



Instruction 3 :

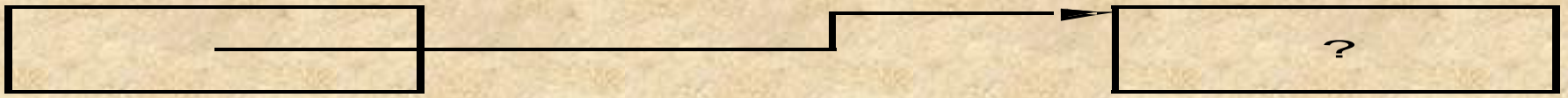
Pt_Integer

Autre_Pt_Integer



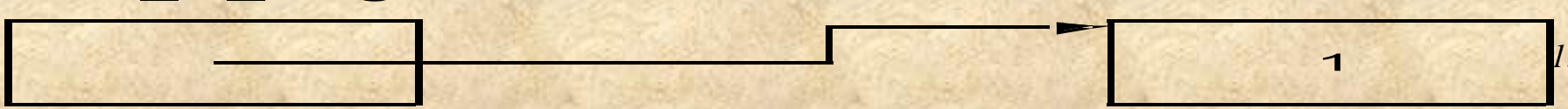
Instruction 5 :

Autre_Pt_Integer



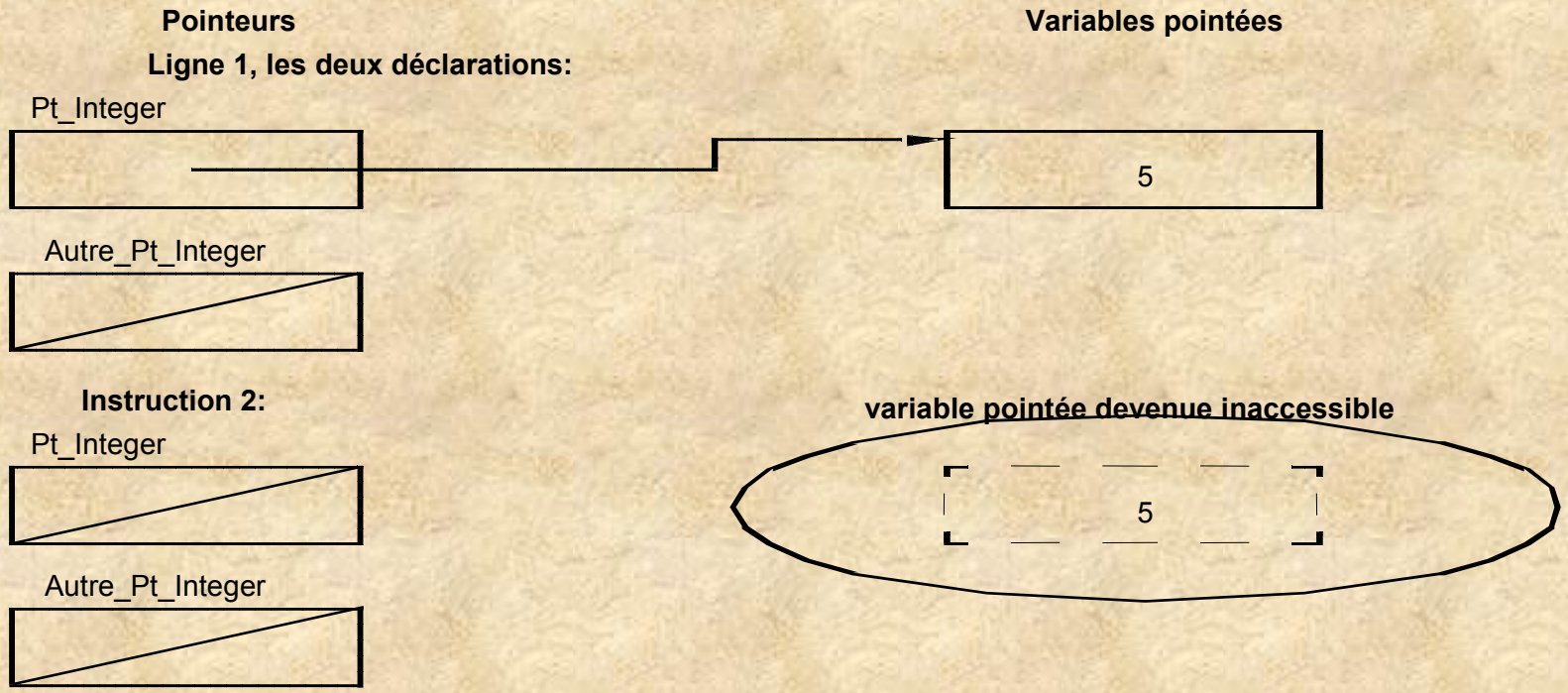
Instruction 6 :

Autre_Pt_Integer



Création de variables inaccessibles

```
type T_Pt_Integer is access Integer;
Pt_Integer : T_Pt_Integer := new Integer'(5);
Autre_Pt_Integer : T_Pt_Integer;          -- 1, situation initiale

Pt_Integer := Autre_Pt_Integer;-- 2
```



Types accès et contraintes

```
subtype T_Intervalle is Integer range 0..100;

-- Discriminant sans valeur par defaut
type T_Article_Sans_Val_Def (Taille : T_Intervalle) is
    record
        Nombre : Integer;
        Tab : String (1..Taille);
    end record;

-- Discriminant avec valeur par defaut
type T_Article_Avec_Val_Def (Taille : T_Intervalle := 10) is
    record
        Nombre : Integer;
        Tab : String (1..Taille);
    end record;
type T_Pt_String is access String;          -- String est non contraint
type T_Pt_Article_Sans_Val_Def is access T_Article_Sans_Val_Def;
type T_Pt_Article_Avec_Val_Def is access T_Article_Avec_Val_Def;
```

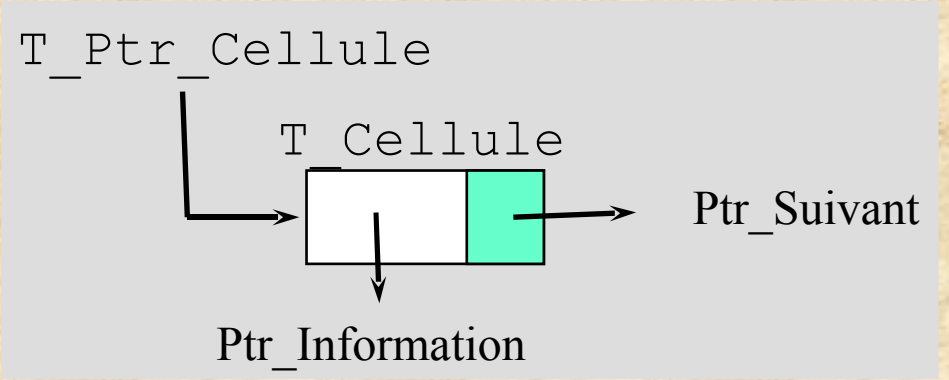
Types accès et contraintes (suite)

```
-- Trois variables pointeurs
Pt_String : T_Pt_String;
Pt_Article_Sans_Val_Def : T_Pt_Article_Sans_Val_Def;
Pt_Article_Avec_Val_Def : T_Pt_Article_Avec_Val_Def;

-- Differents cas d'utilisation de l'allocateur:
-- la contrainte est necessaire
Pt_String := new String (1..10);
Pt_Article_Sans_Val_Def := new T_Article_Sans_Val_Def (10);

-- la contrainte, necessaire, est donnee par la valeur initiale
Pt_String := new String'("Bonjour");            -- Remarquer l'apostrophe
Pt_Article_Sans_Val_Def := new T_Article_Sans_Val_Def'(5, 10, "Hello");

-- la contrainte est possible mais pas necessaire
Pt_Article_Avec_Val_Def := new T_Article_Avec_Val_Def (10);

-- pas de contrainte, mais le discriminant ne pourra cependant pas varier
Pt_Article_Avec_Val_Def:=new T_Article_Avec_Val_Def;
Pt_Article_Avec_Val_Def:=new T_Article_Avec_Val_Def '(5,10,"Hello");
```

Utilisation générale des types accès

Pointeurs sur les objets

```
type T_Cellule;
type T_Ptr_Cellule is access T_Cellule;

type T_Information;
type T_Ptr_Information is access T_Information;
```




```
type T_Cellule is
record
   Ptr_Information : T_Ptr_Information;
   Ptr_Suivant : T_Ptr_Cellule;
end record;

La_Tete, La_queue : T_Ptr_Cellule;
```

Pointeur et variable pointée

Attention à ne pas confondre pointeur et variable pointée (variable dynamique)

type Ptr_Int **is access** Integer;

P1, P2 : Ptr_Int;
P1 := **new** Integer'(23) ;
P2 := P1;

P1, P2 : Ptr_Int;
P1 := **new** Integer'(23) ;
P2 := **new** integer;
P2.all := P1.all;

test d'égalite
P1, P2 : Ptr_Int;
P1 = P2
vaut vrai si P1 et P2 désigne la même variable dynamique ou si valent tous les deux **null**.

P1.all = P2.all
vaut vrai si P1 et P2 désignent des variables dynamiques égales

Libération de l'espace mémoire

```
with Ada.Unchecked_Deallocation;

procedure Liberer_Int is new Ada.Unchecked_Deallocation(Integer, Ptr_Int);
Liberer_Int(P) ;
```

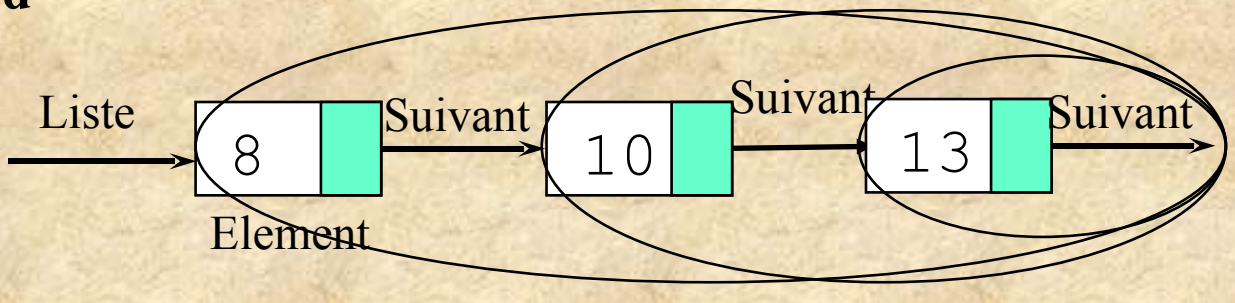
Remarque : Après appel à Liberer_Int : P vaut null

```
type Ptr_Int is access Integer;

P1, P2 : Ptr_Int;
P1 := new Integer'(68);
P2 := new Integer'(35);

P2 := P1;                    ➔ que se passe-t-il ?
Liberer_Int(P1);             ➔ que se passe-t-il ?
```

Liste

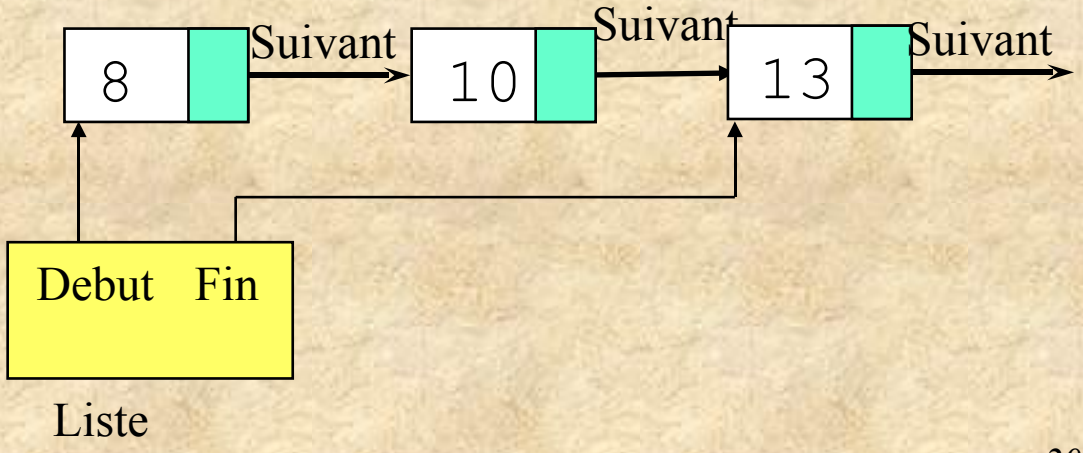
Exemple 1 :

```
type Element;
type Liste is access Element;

type Element is record
   Info : Integer;
   Suivant : Liste;
end record;
```



Exemple 2 :

```
type Element;
type Lien is access Element;
type Element is record
   Info : Integer;
   Suivant : Lien;
end record;

type Liste is record
   Debut : Lien;
   Fin : Lien;
end record;
```

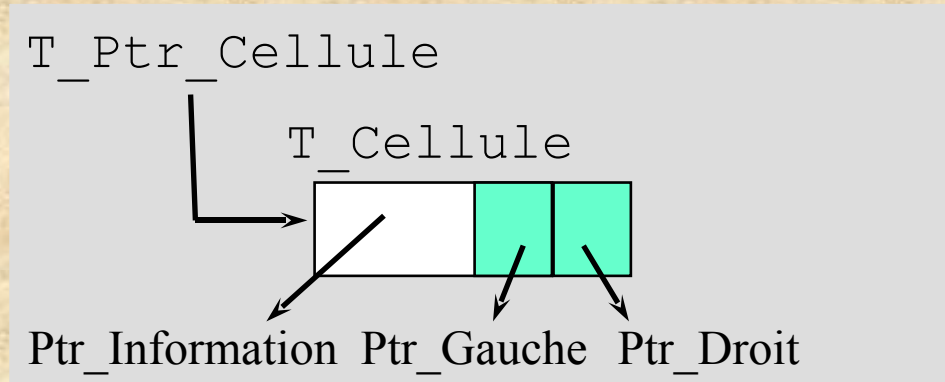


Algorithmes récursifs : exemple parcourir

```
procedure Parcours_Recursif (Deb : in Liste) is
begin
      if Deb /= null then
         Traiter (Deb.All.Nom);
          Parcours_Recursif (Deb.All.Suiv);
      end if;
end Parcours_Recursif;

procedure Parcours_Recursif_Inverse (Deb : in  Liste) is
begin
   if Deb /= null then
       Parcours_Recursif_Inverse(Deb.all.suiv);
       Traiter (Deb.all.nom);
   end if;
end Parcours_Recursif_Inverse;
```

Arbre binaire

```
type T_Cellule;
type T_Ptr_Cellule is access T_Cellule;

type T_Information;
type T_Ptr_Information is access T_Information;
```




```
type T_Cellule is
record
   Ptr_Information : T_Ptr_Information;
   Ptr_Gauche : T_Ptr_Cellule;
   Ptr_Droit  : T_Ptr_Cellule;
end record;

La_Tete, La_queue : T_Ptr_Cellule;
```