Gestion dynamique de la mémoire

Allocations de mémoire

- 2 types d'allocations de mémoire :
  - statique
    - Celle que nous avons utilisée jusqu'à présent
  - dynamique
    - Celle que nous allons voir maintenant

Allocation statique

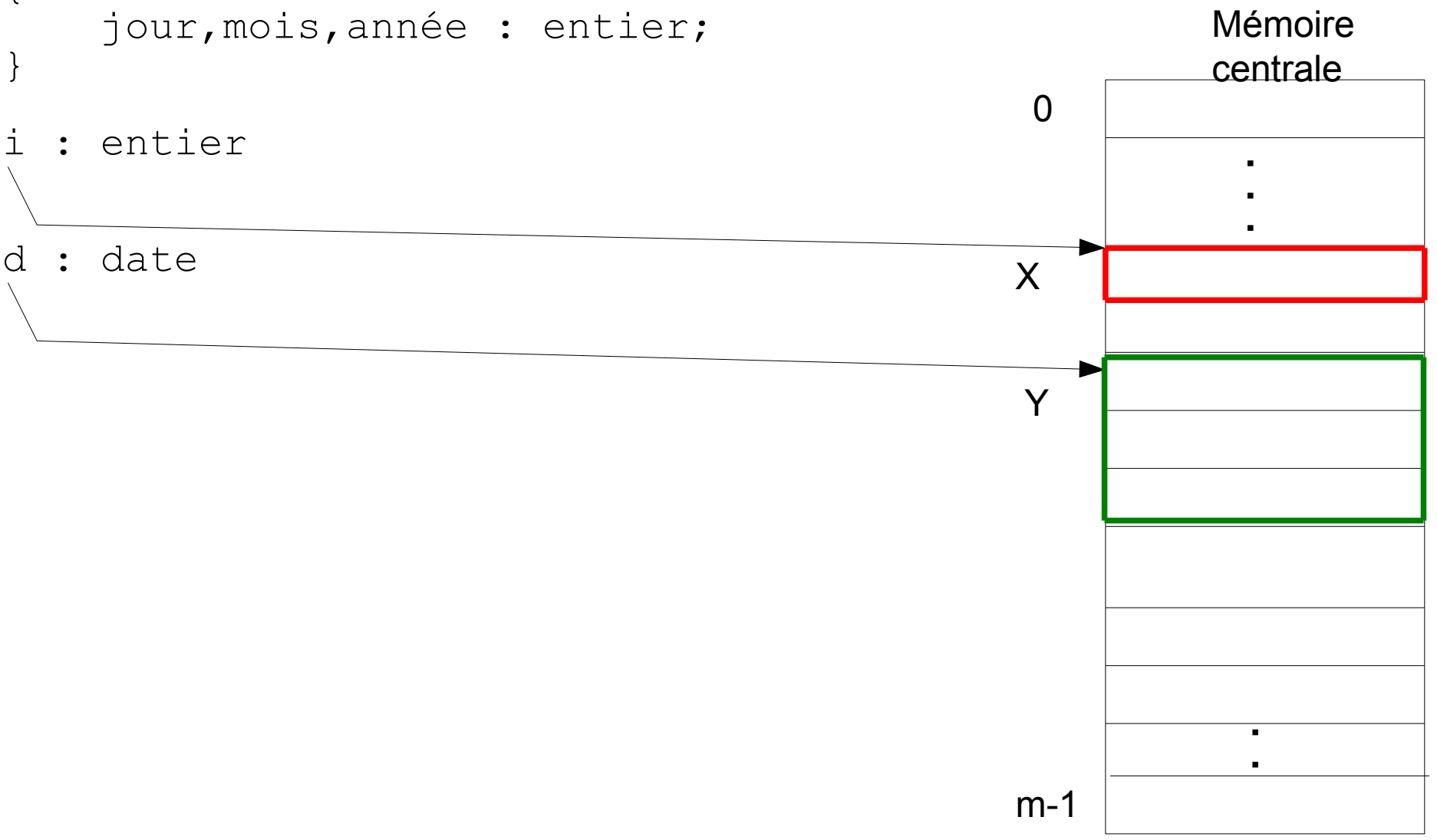
- Dans les algorithmes que nous avons écrits jusqu'à présent, les variables que nous avons déclarées sont :
  - Soit des variables globales (définies dans l'algorithme principal)
  - Soit des variables locales (définies dans les sous-programmes)
  - Ces variables doivent être déclarées explicitement pendant l'écriture des algorithmes et ont une durée de vie précise (la durée de vie du bloc contenant la déclaration) :
    - Les variables globales sont créées au début de l'exécution et sont conservées jusqu'à la fin de l'exécution
    - Les variables locales sont créées au début de l'exécution du sous-programme et sont supprimées à la fin de son exécution

Allocation statique

- **Remarque 1** : Comme nous l'avons vu lors de l'introduction du concept de variable, l'opération de « création » de l'espace mémoire associé à une variable s'appelle l'allocation.
- **Remarque 2** : L'allocation est dite « statique » car décidée au moment de l'écriture de l'algorithme et non évolutive lors de l'exécution.
- Conséquence : Cela a de nombreux inconvénients. Par exemple, il n'est pas possible de contrôler au mieux l'espace mémoire disponible.

Allocation statique

```
structure date
{
    jour,mois,année : entier;
}

i : entier

d : date
```



Allocation statique
Inconvénient 1 : durée de vie

- Si pour des variables simples (entiers, réels, ...) il n'est pas très pénalisant de garder des variables jusqu'à la fin de l'exécution (même si elle ne nous servent plus), il n'en est pas toujours ainsi.
  - Par exemple, imaginons que une variable qui ne soit pas non plus un simple entier mais un vecteur de 200000 éléments entier ! **Pendant l'exécution du programme** (algorithme), il est indispensable de pouvoir supprimer le vecteur s'il n'est plus utilisé et ainsi libérer de la mémoire afin d'éviter une éventuelle saturation.

Allocation statique Inconvénient 2

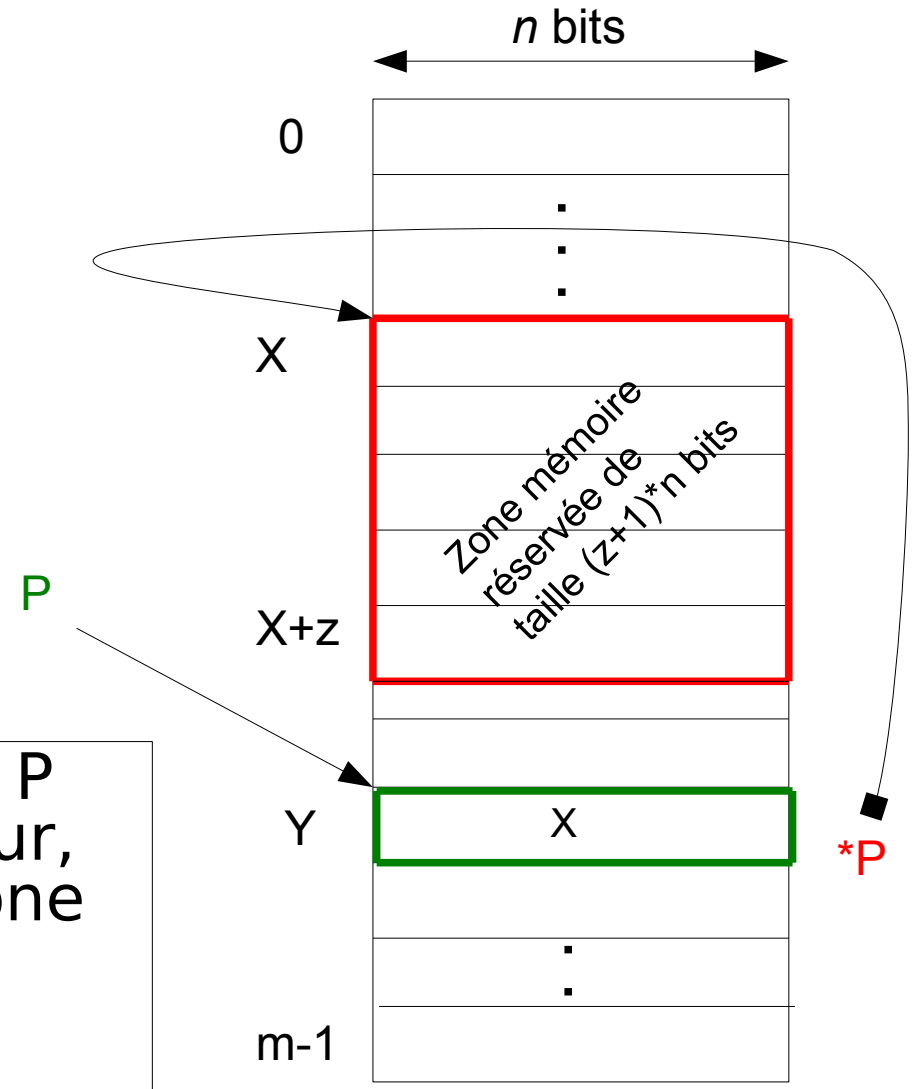
- Fréquemment, par exemple dans le cas d'un vecteur, on ne peut pas connaître, a priori lors de l'écriture de l'algorithme, le nombre d'éléments qu'il contiendra. Cette taille n'est bien souvent calculable ou connu **que pendant** l'exécution du programme.
- Il est donc indispensable de disposer d'un mécanisme permettant de créer (et de supprimer) des variables de différents types pendant l'exécution.
- C'est ce qu'on appelle **la gestion dynamique** (pendant l'exécution) de la mémoire. Elle comprend deux opérations:
  - Allouer
  - Libérer

Allocation dynamique

- Il n'y a pas de déclaration explicite de variables.
- L'accès se fait par le moyen d'un pointeur (1 adresse mémoire)
- L'espace mémoire existe juste pour la durée nécessaire.
- On utilise l'opérateur `Allouer`
  - Cette opération permet de réserver une zone mémoire et permet d'accéder à cette zone en utilisant la valeur qu'elle retourne
  - On peut stocker cette valeur, une adresse mémoire, dans ce que l'on appelle **un pointeur**.

Rappel sur la notion de pointeur

- Un pointeur est une variable de type `pointeur` qui :
  - a un nom
  - pointe une zone mémoire (désignée par une adresse) qui doit avoir été réservée préalablement à son utilisation



Ici la variable P est un pointeur, pointant la zone mémoire à l'adresse X.

Syntaxe pour les pointeurs

- Déclaration d'un pointeur :

```
nom_pointeur : pointeur sur type_pointé
```

- Allocation mémoire d'un pointeur

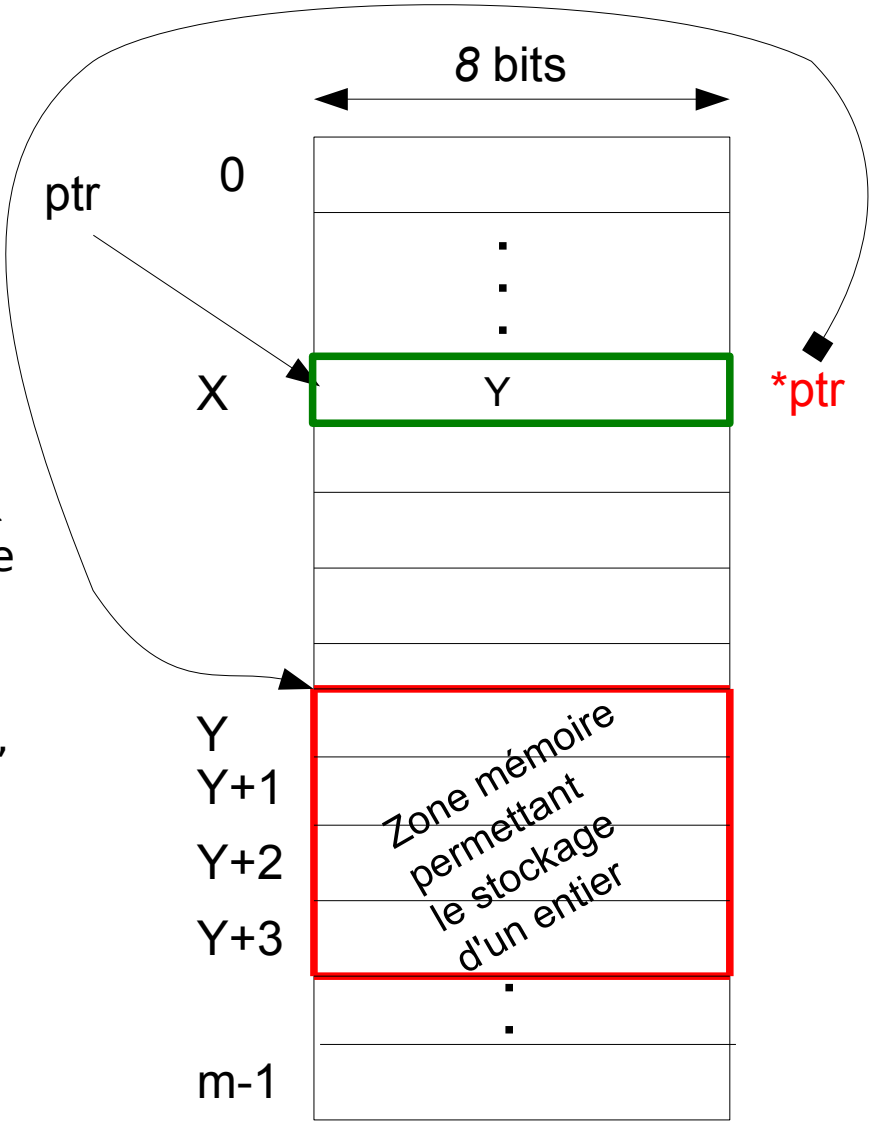
```
nom_pointeur ← Allouer type_pointé
```

- Dans l'algorithme, afin de pouvoir utiliser un pointeur, il doit obligatoirement y avoir
  - tout d'abord, la déclaration (qui permet la création d'une zone en mémoire qui pourra contenir une adresse (la zone désignée par la variable P sur le schéma précédent)
  - puis l'allocation qui permet de créer la zone mémoire correspondant au type_pointé (zone en rouge sur le schéma précédent) et affecte l'adresse (X sur le schéma précédent) comme valeur à la variable de type `pointeur`.

Exemple de pointeur

```
Algorithme Exemple_de_pointeur

{

  variable ptr : pointeur sur entier

  ptr ← Allouer entier

  ...

}
```

Pour simplifier, ici, on suppose une mémoire 8 bits et contenant m emplacements, où m = 256. Ainsi, une adresse se représente sur 1 seul emplacement mémoire (sur 8 bits).



`ptr` est une variable de type `pointeur sur entier`. Il occupe donc un espace lui permettant de contenir une adresse en fonction de l'architecture utilisée. Ainsi dans le monde réel sur une architecture 32 bits le pointeur devrait occuper 32 bits (soit la même taille qu'un entier sur 32 bits – ce qui dans le cadre de notre exemple est un peu idiot ! Bref ...).

La déclaration permet de réserver la zone verte. À ce moment là, la variable `ptr`, de type `pointeur sur entier`, contient une valeur par particulière par défaut qui s'appellera `NULL`. Cette valeur indique que le pointeur pointe sur un espace vide (il n'y a pas eu d'allocation)

Une fois que l'instruction `Allouer entier` a eu lieu, un espace mémoire permettant le stockage d'un entier a été créé (désigné par son adresse de début X et représenté en rouge) et puisque le résultat de cette opération est affectée (←) au pointeur `ptr`, celui-ci contient l'adresse de la zone (soit Y ici). il est possible d'accéder à l'espace mémoire permettant le stockage d'un entier (désigné par son adresse de début X et représenté en rouge) via l'opérateur de déréférencement `*ptr`

Syntaxe pour les pointeurs

- Désallocation d'un élément pointé

```
Libérer nom_pointeur
```

- Cette opération libère la zone mémoire occupée par l'élément pointé et affecte `NULL` comme valeur au pointeur.
- Toutefois, attention dans la pratique, selon le langage de programmation utilisé, lorsque la zone pointée est désallouée, le pointeur conserve sa valeur et par conséquent, il pointe sur une zone mémoire non valide.
  - Or, il ne faut jamais accéder au contenu d'une zone mémoire libérée sous peine de causer un problème à l'exécution (en effet la zone a pu être réutilisée entre temps par un autre programme).
  - La bonne pratique veut que dès que l'on a désalloué un pointeur, on lui donne la valeur particulière NULL (on peut aussi le faire en algorithmique)

Manipulation de pointeur

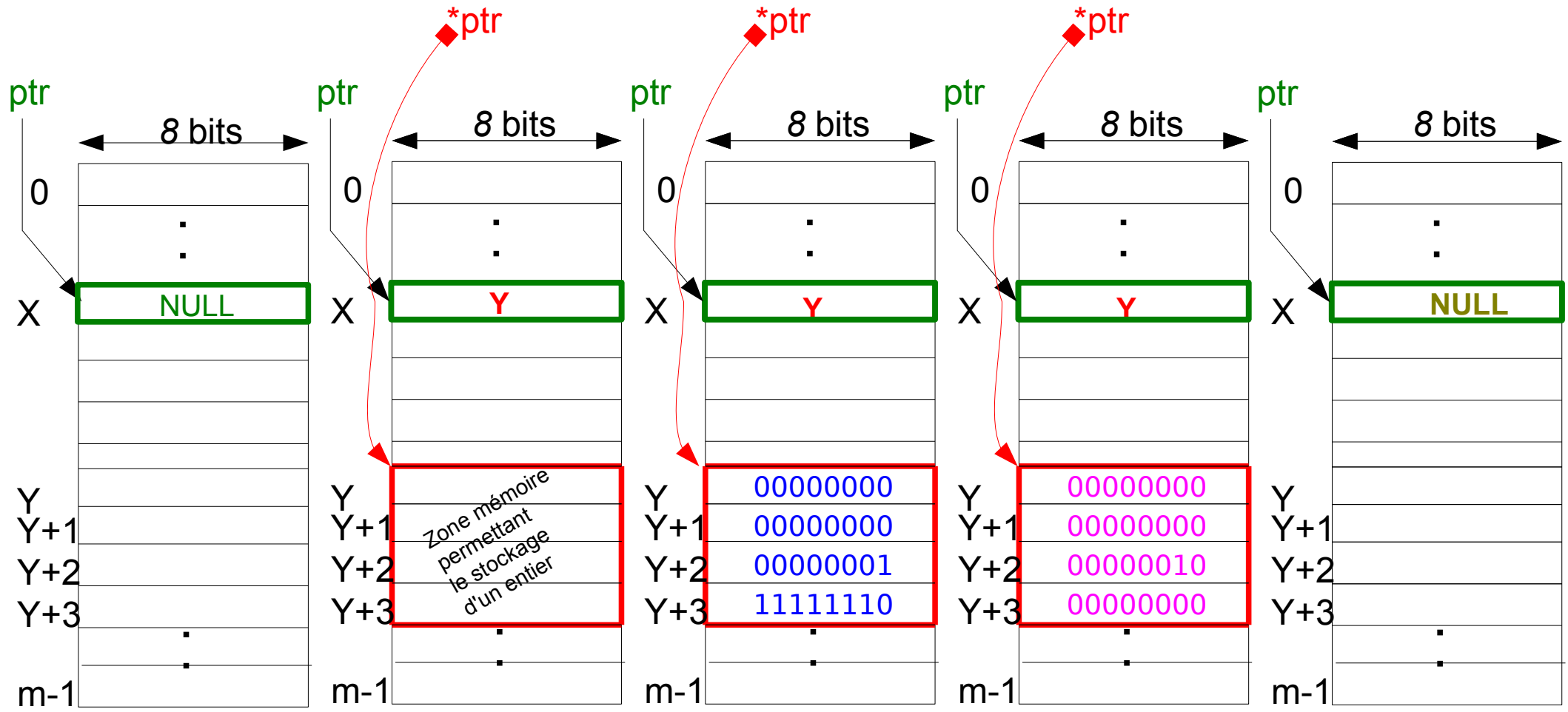
- Si un pointeur permet de désigner un espace mémoire, il est utile de pouvoir modifier cet espace mémoire. Pour cela l'opérateur unaire * (opérateur de déréférencement) permet d'accéder à la zone mémoire pointée comme s'il s'agissait d'une variable.
- Rappel :
  - L'opérateur * permet donc d'accéder au contenu de l'adresse.
  - L'opérateur * s'applique à l'expression (en général, il s'agit d'une variable) qui le suit.
- Exemple :

```
*nom_pointeur ← une_valeur_compatible_avec_le_type_pointé
```

Exemple de pointeur

```
Algorithme Exemple_de_pointeur_2

{
   variable ptr : pointeur sur entier
   ptr ← Allouer entier
   *ptr ← 510
   *ptr ← *ptr + 514
   Libérer ptr

}
```

Pour simplifier, ici, on suppose une mémoire 8 bits et contenant m emplacements, où m = 256. Ainsi, une adresse se représente sur 1 seul emplacement mémoire (sur 8 bits).

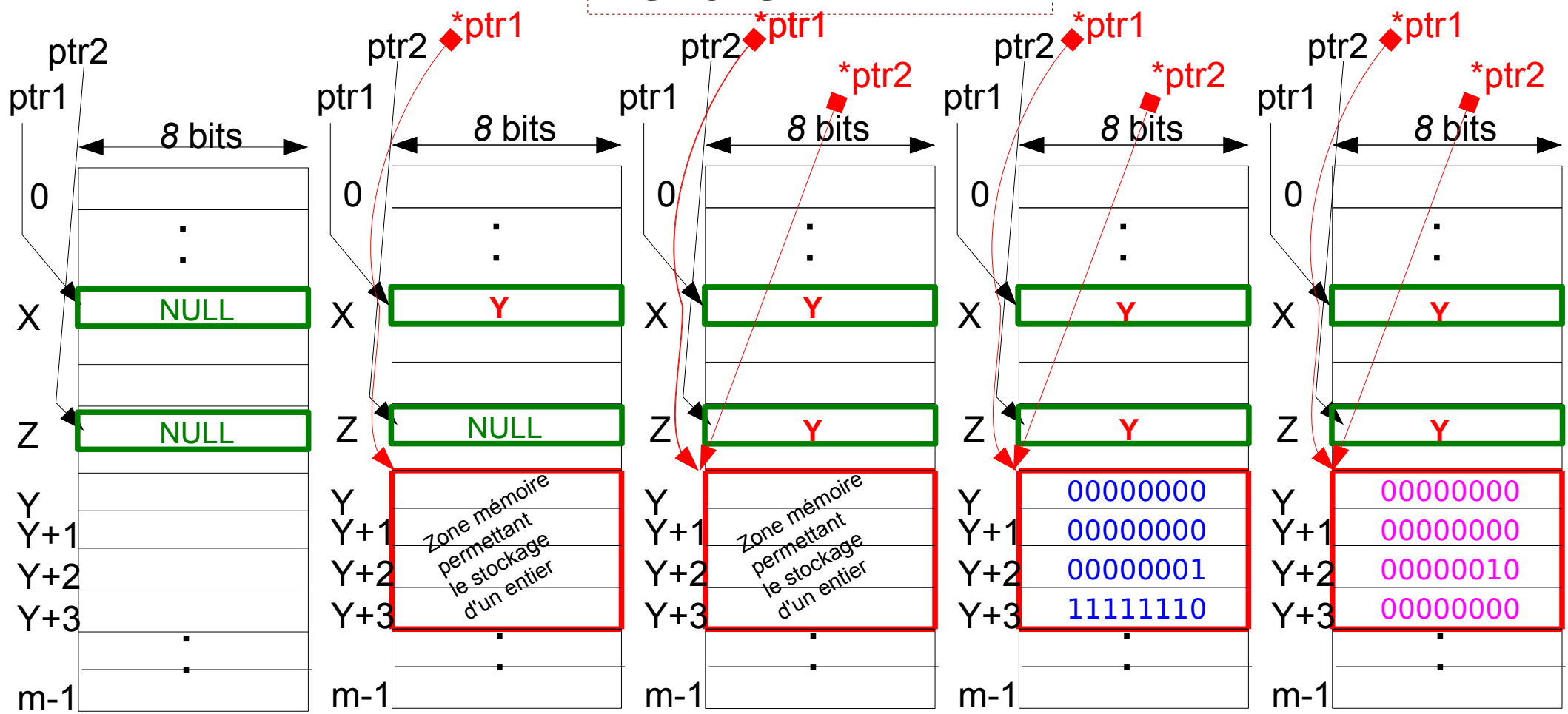


Exemple de pointeur

```
Algorithme Exemple_de_pointeur_3

{
   variable ptr1, ptr2 : pointeur sur entier
   ptr1 ← Allouer entier
   ptr2 ← ptr1
   *ptr1 ← 510
   *ptr2 ← *ptr2 + 514

}
```

Pour simplifier, ici, on suppose une mémoire 8 bits et contenant m emplacements, où m = 256. Ainsi, une adresse se représente sur 1 seul emplacement mémoire (sur 8 bits).

On peut affecter un pointeur à un autre. Deux pointeurs peuvent désigner et utiliser la même zone mémoire.



Pointeur : des erreurs à éviter

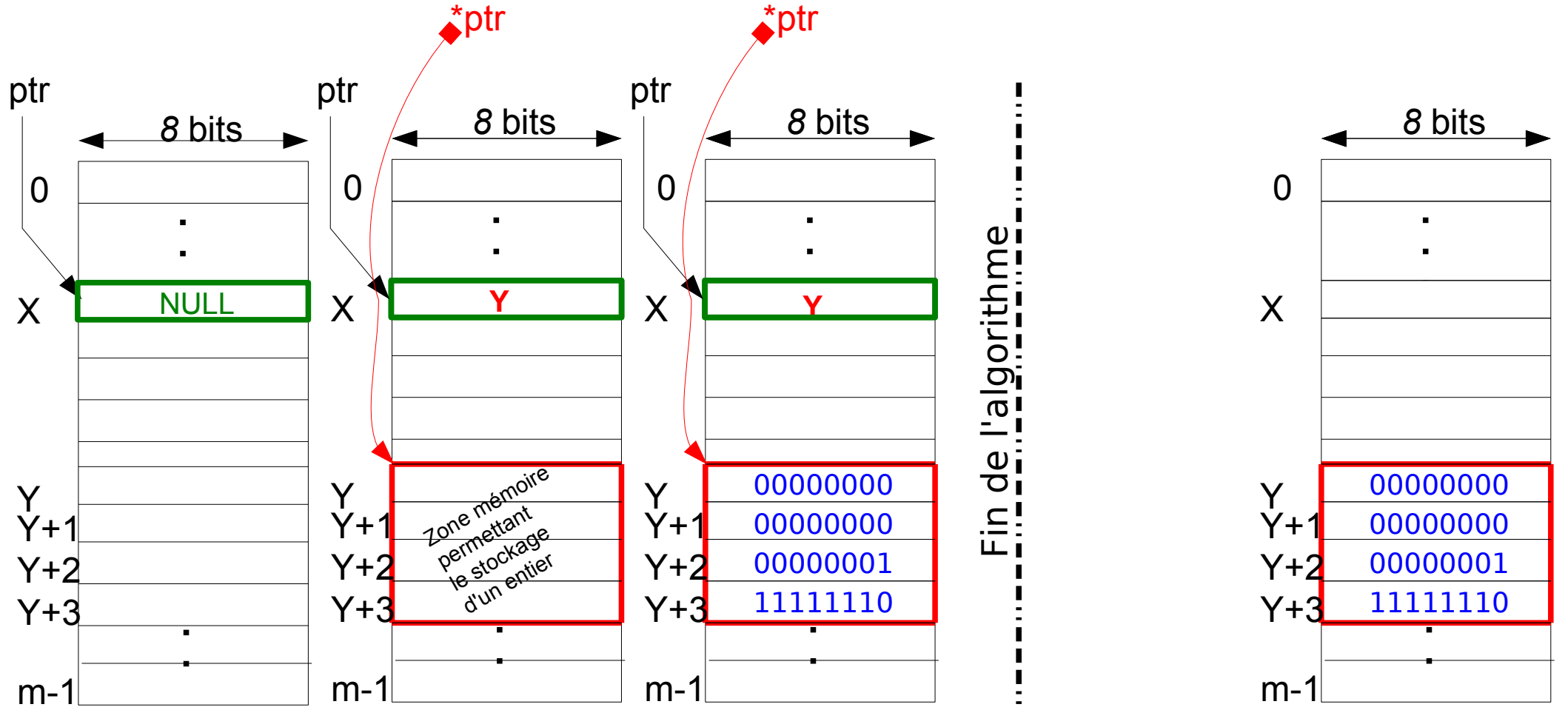
- Une première erreur dans l'utilisation des pointeurs, consiste à oublier de désallouer l'espace mémoire avant la fin de l'algorithme/programme (auquel cas l'espace mémoire continue à être réservé et donc ne peut pas être utilisé par les autres algorithmes/programmes en cours d'exécution ou allant s'exécuter dans l'avenir.

Illustration de l'erreur 1

```
Algorithme Exemple_de_pointeur_4

{
   variable ptr : pointeur sur entier
   ptr ← Allouer entier
   *ptr ← 510

}
```

Pour simplifier, ici, on suppose une mémoire 8 bits et contenant m emplacements, où m = 256. Ainsi, une adresse se représente sur 1 seul emplacement mémoire (sur 8 bits).

Après la fin de l'exécution de l'algorithme, l'entier alloué continue à occuper inutilement de la mémoire.
Cette zone ne sera plus accessible pour aucun algorithme/programme. La mémoire est « perdue ».



Pointeur : des erreurs à éviter

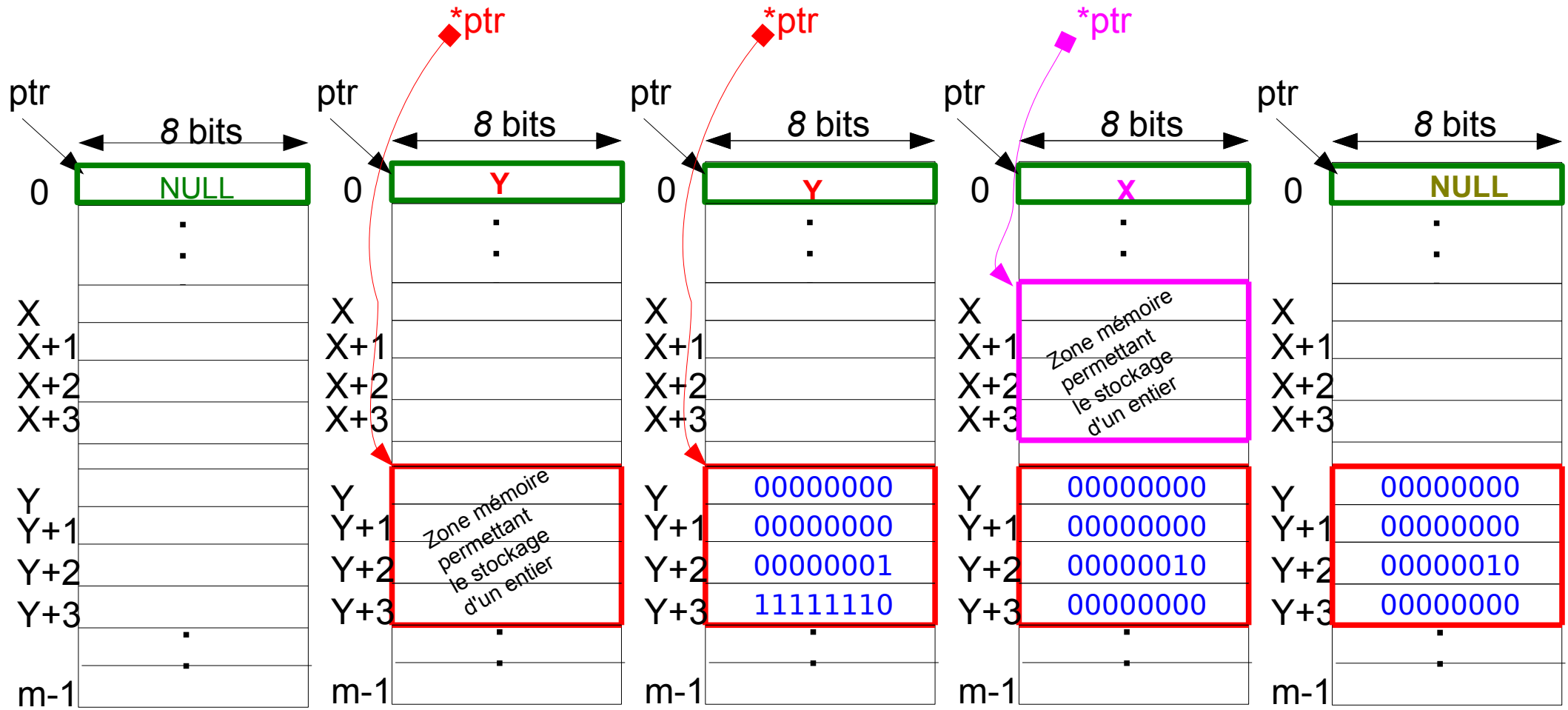
- Une seconde erreur consiste à allouer un nouvel espace mémoire et à l'assigner à un pointeur en oubliant de désallouer l'espace mémoire précédemment pointé par ce dernier (ce qui est problématique si aucun autre pointeur ne désigne cet espace – en effet cet espace est désormais considéré comme « perdu »)

Illustration de l'erreur 2

```
Algorithme Exemple_de_pointeur_5

{
   variable ptr : pointeur sur entier
   ptr ← Allouer entier
   *ptr ← 510
   ptr ← Allouer entier
   Libérer ptr

}
```

Pour simplifier, ici, on suppose une mémoire 8 bits et contenant m emplacements, où m = 256. Ainsi, une adresse se représente sur 1 seul emplacement mémoire (sur 8 bits).

Après la seconde allocation effectuée au pointeur, l'espace mémoire rouge est « perdu ». Plus aucun pointeur ne connaissant son adresse pour y accéder, il n'existe plus de moyen de libérer cette zone. Il aurait fallu copier le pointeur dans un autre, préalablement à la nouvelle allocation afin de toujours pointer cette zone.



Pointeur : des erreurs à éviter

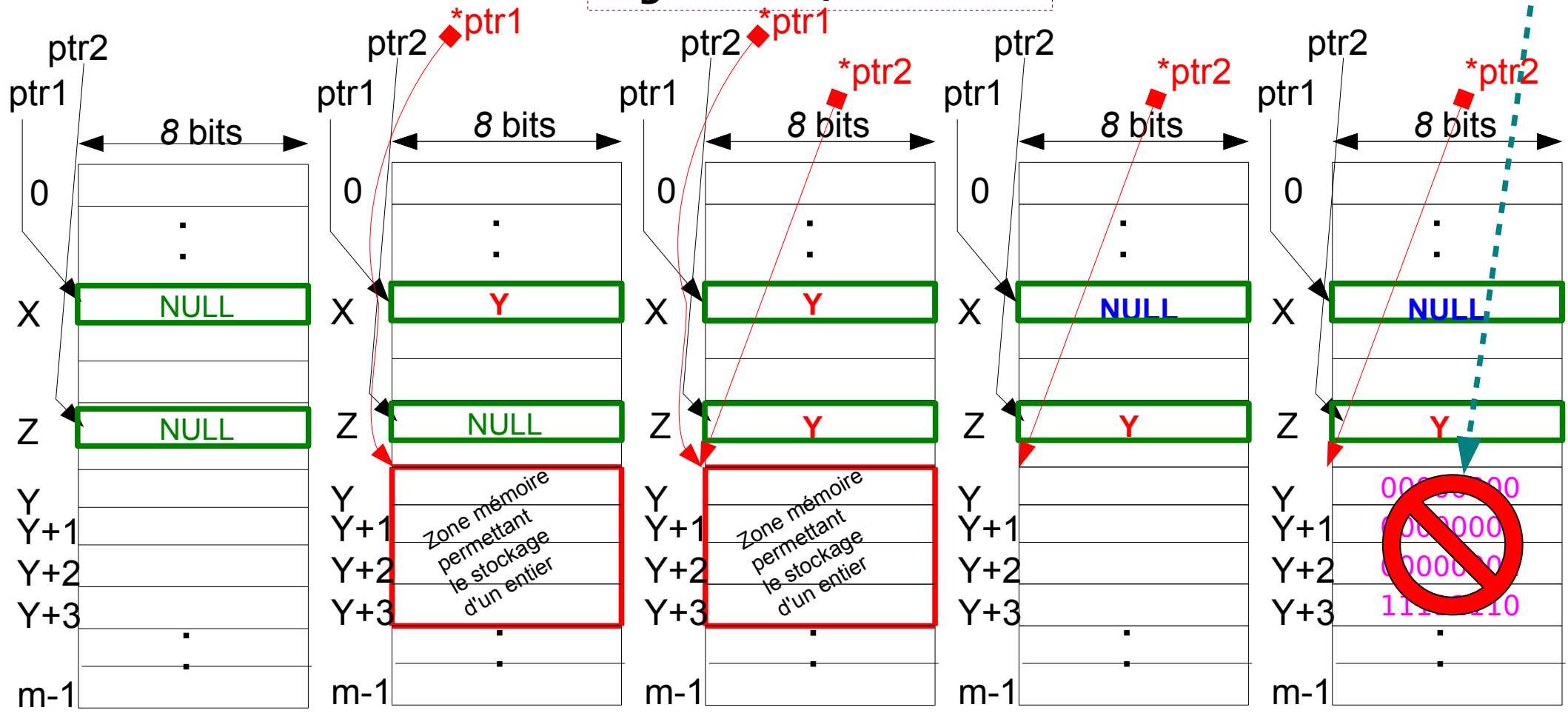
- Une troisième erreur est de considérer que ce n'est pas parce qu'un pointeur n'est pas NULL qu'il pointe sur une zone valide.
- Illustration sur le transparent suivant !
- Bonne pratique : mettre à NULL tous les pointeurs désignant la même zone mémoire dès que l'un d'eux est désalloué.

Illustration de l'erreur 3

```
Algorithme Exemple_de_pointeur_6

{
   variable ptr1, ptr2 : pointeur sur entier
   ptr1 ← Allouer entier
   ptr2 ← ptr1
   Libérer ptr1
   *ptr2 ← 510

}
```

Impossible puisque la zone mémoire n'est plus réservée (elle peut donc être utilisée par un autre algorithme).

Pour simplifier, ici, on suppose une mémoire 8 bits et contenant m emplacements, où m = 256. Ainsi, une adresse se représente sur 1 seul emplacement mémoire (sur 8 bits).



Remarque sur les pointeurs

- Il est impossible d'affecter directement une valeur à un pointeur

```
variable ptr : pointeur sur entier
ptr ← Allouer entier
ptr ← 10
```

- En revanche l'affectation suivante est valide :

```
*ptr ← 10
```

- Il n'est pas possible d'afficher une variable pointeur

```
Afficher(ptr)
```

- En revanche il est possible d'afficher le contenu d'une variable pointeur

```
Afficher(*ptr)
```

Exemple de bonnes pratiques

```
Algorithme Un_Exemple_Pour_Finir_Avec_Les_Pointeurs_Illustrant_De_Bonnes_Pratiques

{

  variable ptr : pointeur sur entier

  ptr ← Allouer entier

  Afficher("Saisissez un entier :")

  Saisir(*ptr)

  Afficher("Vous avez saisi :", *ptr)

  Libérer ptr

  ptr ← NULL

  ...

  Si (ptr = NULL) alors

  {

      ...

      ptr ← Allouer entier

      ...

  }

  ...

  Si (ptr ≠ NULL) alors

  {

      ...

      Libérer ptr

      ...

  }

}
```

Exemple de bonne pratique.
Libérer dès qu'on n'a plus besoin de l'espace mémoire et mettre à NULL le pointeur

Exemple de bonne pratique.
Tester le pointeur avant de le réallouer

Exemple de bonne pratique.
Tester le pointeur avant de le libérer

Allocation dynamique et vecteur

- Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter un vecteur. Bien sûr, une solution consisterait à déclarer un vecteur d'une taille très importante, mais engendrerait beaucoup de problèmes.
- Aussi, pour parer à cette situation, nous avons la possibilité de déclarer le vecteur sans préciser au départ son nombre d'éléments (souvenez vous certains passages de vecteur en paramètre de fonction ou procédure).
- Ce n'est que dans un second temps, au cours de l'algorithme, que l'on va fixer ce nombre via l'opérateur `Allouer`
- Un vecteur sans taille initiale se déclare ainsi :

```
V: vecteur d'entier
```

Exemple de déclaration dynamique

```
Algorithme Exemple_Declaration_Dynamique_vecteur_Et_Tri

{

  variable tab1, tab2, tab_fusion : vecteur d'entier
           n1, n2, n3 : entier


  Saisir_tab(tab1, n1) //La procédure Saisir_tab est donnée après
  Saisir_tab(tab2, n2)

  // Tri les deux vecteurs avec la procédure vue dans le cours
  Tri_min_max(tab1, n1)
  Tri_min_max(tab2, n2)

  n3 ← n1 + n2
  tab_fusion ← Allouer vecteur de n3 entier

  // Appel à la procédure de fusion vue dans le cours.
  Fusion_2vecteurs(tab1, tab2, n1, n2, tab_fusion)

  Libérer tab1
  tab1 ← NULL

  Libérer tab2
  tab2 ← NULL

  // Faire ici un affichage de tab_fusion

  Libérer tab_fusion
  tab_fusion ← NULL

}
```

L'allocation dynamique peut avoir lieu dans un sous programme et la libération intervenir dans l'algorithme principal (et vice versa). Il faut juste faire attention à ce que l'on fait.

Exemple de déclaration dynamique

```
procedure Saisir_tab(E/S V: vecteur d'entier, S N: entier)
{
    variable i: entier
    Afficher("Saisissez le nombre d'éléments à saisir")
    Saisir(N)
    V ← Allouer vecteur de N entier
    Pour i de 0 à N-1
    {
        Afficher("Saisir la valeur de l'élément ", i)
        Saisir(V[i])
    }
}
```

Pointeur et enregistrements

- Il est bien sur possible de déclarer un pointeur sur un type structuré. Ainsi l'accès au champ se fera de la façon suivante :

```
(*nom_pointeur).champ_de_l_enregistrement
```

- Toutefois pour des raisons de commodité, une autre syntaxe existe.

```
nom_pointeur→champ_de_l_enregistrement
```

Exemple : Pointeur et enregistrement

```
structure Pays
{
     nom : chaine de caractère
     nb_hab : entier
}

Algorithme Pointeur_Enregistrement
{
     variable ppays : pointeur sur Pays
     ppays ← Allouer Pays
     Afficher("Saisissez le nom du pays et sa population")
     Saisir(ppays→nom,ppays→nb_hab) // équivalent de
                                    // Saisir((*ppays).nom,(*ppays).nb_hab)
     Afficher("Vous avez saisi que le ", ppays→nom," a ", ppays→nb_hab,"habitant(s).")
     Libérer ppays
}
```

Exemple : Vecteur, allocation dynamique, enregistrement, tri

- Écrire l'algorithme demandant à l'utilisateur de saisir le nombre de pays membres de l'ONU, de saisir les noms et nombre d'habitants de chacun d'eux puis qui triera ces pays dans l'ordre décroissant du nombre d'habitants avant des les afficher.

```
structure Pays
{
    nom : chaine de caractère
    nb_hab : entier
}

Algorithme Tri_Pays_ONU
{
    variable v_pays : vecteur de Pays
             nb_pays :

    Saisie_Pays_ONU(v_pays, nb_pays)

    Tri_Pays_par_nb_hab(v_pays, nb_pays)

    Afficher_Pays(v_pays, nb_pays)

    Libérer v_pays
    v_pays ← NULL
}

procédure Saisie_Pays_ONU(E/S v:vecteur de Pays, S n:entier)
{
    variable i : entier

    Afficher("Saisissez le nombre de pays actuellement membre de l'ONU :")
    Saisir(n)

    v ← Allouer vecteur de n Pays
    Pour i de 0 à n-1
    {
        Afficher("Saisir le nom du pays :")
        Saisir(v[i].nom)
        Afficher("Saisir le nombre d'habitants du pays :")
        Saisir(v[i].nb_hab)
    }
}
```

```
procédure Tri_Pays_par_nb_hab(E/S v:vecteur de Pays, E n:entier)
{
    variable i, etape, i_min: entier
             p: Pays
    Pour etape de 1 à (n - 1)
    {
        /* Recherche du min */
      i_min ← 0
      Pour i de 1 à (n - etape)
            Si (v[i].nb_hab < v[i_min].nb_hab) alors
                i_min ← i
        /* permutation */
        p ← v[i_min]
        v[i_min] ← v[n - etape]
        v[n - etape] ← p
    }
}

procédure Afficher_Pays(E/S v:vecteur de Pays, E n:entier)
{
    variable i : entier

    Afficher("La liste des ",n," pays membres de l'ONU classés par l'importance de leur
population sont : ")

    Pour i de 0 à n-1
    {
        Afficher(i+1,") ",v[i].nom," avec ",v[i].nb_hab," habitants.\n")
    }
}
```

Adaptation du tri par recherche du minimum.

Allocation dynamique en C++

- Pour une allocation dynamique l'opérateur `new` est utilisé.
- Exemples :

```
int *p; // déclaration d'un pointeur sur 1 entier
p = new int; // allocation de mémoire pour 1 entier
```

ou directement : `int *p = new int;`

```
float *tab;
tab = new float[30]; // allocation de mémoire pour 30 réels
```

ou directement : `float *tab = new float[30];`

```
char *ch;
ch = new char[30]; // allocation de mémoire pour 30 caractères
```

ou directement : `char *ch = new char[30];`

Désallocation en C++

- Pour libérer physiquement la place allouée l'opérateur `delete` est utilisé.
- Exemples :

```
delete p; // libérer la place allouée par new int
```

=> `p` reçoit `NULL`

```
delete tab;
delete ch;
```

Pointeur sur structure en C++

- L'accès au champ se fera de la façon suivante :

```
(*nom_pointeur).champ_de_l_enregistrement
```

- Ou avec une syntaxe plus commode, une autre syntaxe existe.

```
nom_pointeur->champ_de_l_enregistrement
```

- Exemple

```
struct point
{
float x,y;
};

...

point *pp = new point;
*pp.x = 12;                    // première façon
pp->y=7;                       // deuxième façon
```