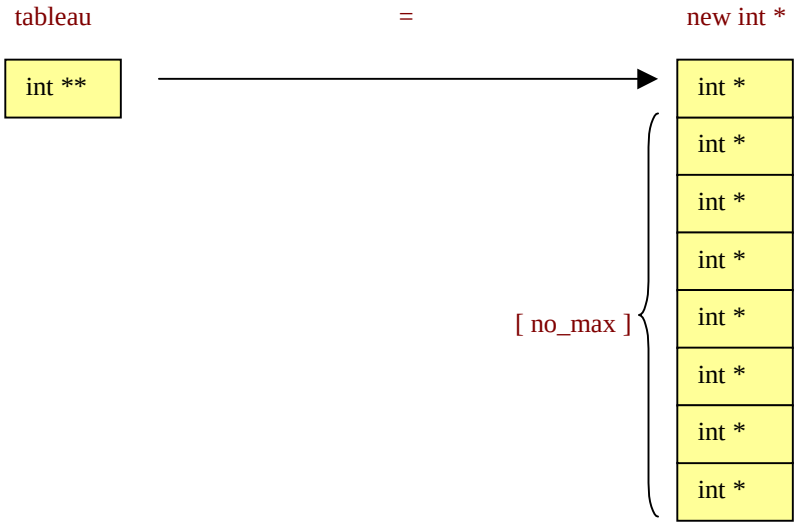
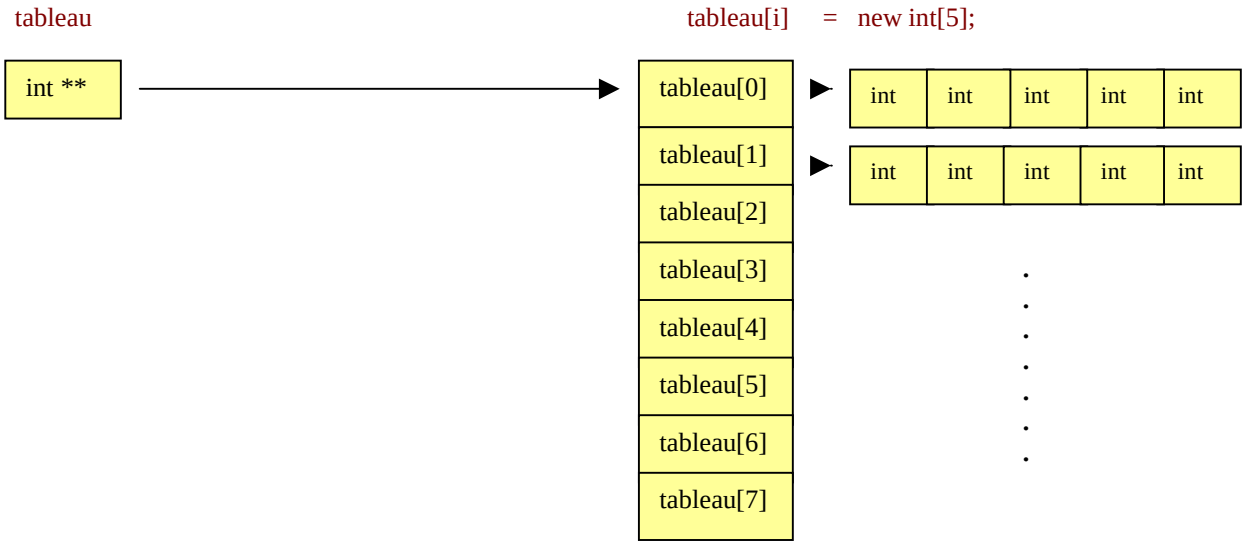
Allocation dynamique en c++

int **tableau;

tableau = new int *[no_max];

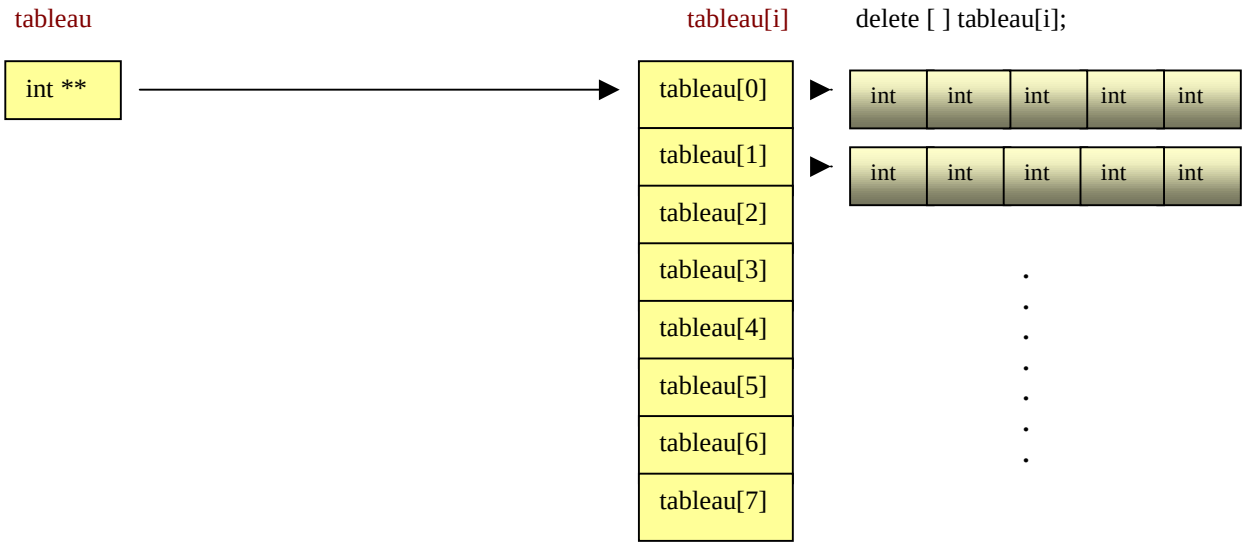


for(int i=0; i<no_max; i++)
** tableau[i] = new int[5];**

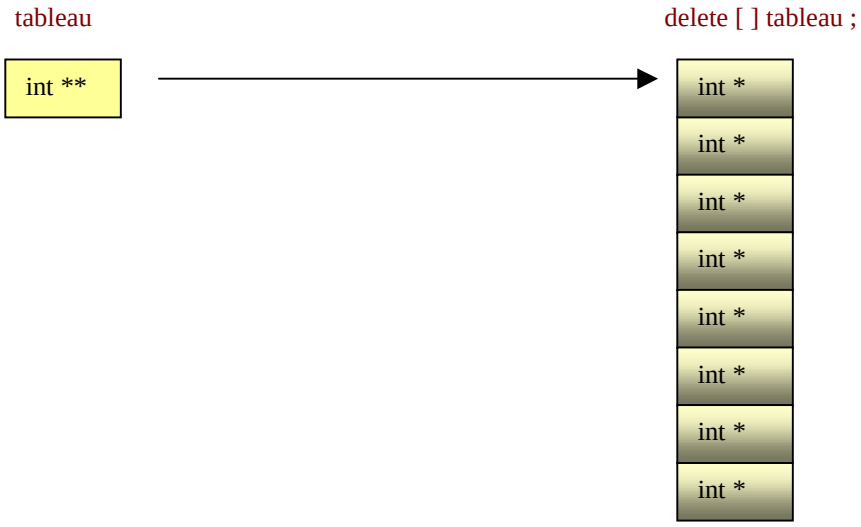


Déallocation en c++

```
for(int i=0; i<no_max; i++)
   delete [ ] tableau[i];
```



delete [ ] tableau ;



À la fin il ne reste que le pointeur initial qui avait été déclaré normalement.

Gérer les erreurs d'allocation de mémoire en c++

La valeur de retour de new est, soit l'adresse de la mémoire allouée en cas de succès, soit zéro en cas d'échec. Il est est important de tester la réussite de l'allocation dynamique pour éviter ultérieurement la référence d'un pointeur null. Dans l'hypothèse qu'une allocation ratée ne sera probablement pas suivie d'une allocation réussie, nous terminons la fonction illico quand un échec survient.

```
int **tableau;
tableau = new int *[ no_max ];
if(tableau == 0)
  return (int **)0;
for(int i=0; i<no_max; i++)
{
  tableau[i] = new int[ 5 ];
  if (tableau[i] == 0)
  {
    for(int j=0; j<i; j++ )
      delete [ ] tableau[ j ];
    delete [ ] tableau;
    return (int **)0;
  }
}
return tableau ;
```

ou encore

```
int **tableau;
if( (tableau = new int *[ no_max ] ) == 0 )
  return (int **)0;
for(int i=0; i<no_max; i++)
{
  if( ( tableau[i] = new int[ 5 ] ) == 0 )
  {
    for(int j=0; j<i; j++ )
      delete [ ] tableau[ j ];
    delete [ ] tableau;
    return (int **)0;
  }
}
return tableau ;
```

Fréquentes erreurs de programmation

La mémoire allouée par new devrait être libérée par delete pour usage par le système d'exploitation aussitôt que l'on en n'a plus besoin. Le défaut de faire cela produira un programme gourmand en mémoire vive. L'appel de l'opérateur delete est relié à l'appel précédent de new pour ce même pointeur : **un delete pour un new**. L'operation delete pointeur déalloue la mémoire pointée si cette mémoire provient d'un new.

En C++, si le new avait alloué plus d'un élément en utilisant les [ ], par exemple *ptr = new int[5]* l'opérateur delete spécifiera qu'il s'agit d'un tableau à effacer avec le symbole [ ] immédiatement après delete *delete [ ] ptr ;*

Il est dangereux de faire un delete sur un pointeur qui ne pointe pas à de la mémoire qui vient de new. Par exemple, il ne faut pas faire delete sur un pointeur pointant sur une variable locale automatique. Il ne faut évidemment pas faire delete sur un pointeur s'il a déjà été 'deleté' (pas deux fois de suite, pas deux fois sans que new intervienne entre les deux…). Une dernière erreur, la plus fréquente, est la tentative d'accès à de la mémoire libérée. Une fois que la mémoire est déallouée par delete, il est illégal de référencer le pointeur avec *pointeur ou pointeur[4].

Libération de la mémoire

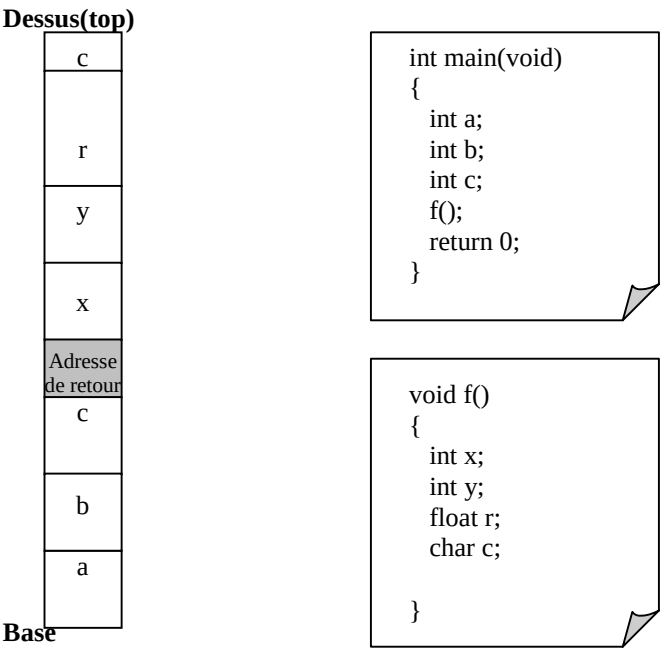
Parce que la zone mémoire utilisée par new et malloc() est une zone commune à tous les programmes, il est important de libérer celle dont on a plus besoin le plus tôt possible pour utilisation par les autres programmes ou par le nôtre ultérieurement.

Bien entendu, il est important de se souvenir de l'adresse de toutes ces portions de mémoire pour pouvoir **toutes** les libérer. Par exemple, dans le code précédent, si le premier pointeur int ** avait été libéré en premier, avant de libérer chacun des pointeurs int * du premier tableau, nous aurions effacé ce tableau de pointeurs et il aurait été ensuite impossible de libérer la mémoire dont ils contiennent l'adresse (souvenons-nous que chacun de ces pointeurs int * mémorisait l'adresse d'un bloc de 5 entiers). Toutes ces adresses auraient été perdues à jamais, donc impossible de retracer leur position et de les redonner au système d'exploitation.

Le principe général de libération de la mémoire est de libérer les positions en **ordre inverse de l'allocation** pour éviter d'effacer l'indice d'un autre bloc de mémoire encore possédé par le programme. Ceci est vrai pour les déallocations de tableaux multidimensionnels ou de toute structure de données maintenue par des tableaux de pointeurs (structure des gros fichiers dans le système Unix par exemple). Ceci est aussi vrai pour les structures de données comme les listes est les arbres maintenus avec des blocs de données qui pointent les uns vers les autres (nœud qui sont capables de référencer d'autres nœud du même type que eux).

La théorie

Nous savons que les variables déclarées dans une fonction sont détruites à la sortie de la fonction, c'est la loi du domaine de validité des identificateurs. Le fonctionnement est très simple, le programme utilise une pile (stack en anglais) où il dépose successivement les variables qu'il crée. La première variable se retrouve donc en-dessous complètement et la dernière créée est sur le dessus. Lors de l'appel d'une fonction, la pile sert aussi à storer où on en était dans le programme avant l'appel de la fonction. Pendant l'exécution de la dite fonction, elle crée ses propres variables qui sont ajoutée sur le dessus de la pile, et lorsque la fonction return, toutes les variables du dessus sont considérées détruites car on retourne se positionner plus bas, là où on était avant l'appel de la fonction. C'est la position du **stack pointer** qui détermine ainsi le dessus de la pile à nos yeux.



Les variables seront donc « détruites » dans l'ordre inverse de leur création, à partir du dessus de la pile jusqu'à la base. Bien sur cela se fait tout au long du programme, une création de piles temporaires sur le dessus de la pile se produit à chaque utilisation des accolades { }. Ainsi, une variable créée dans un bloc d'instructions attaché à un if devrait être détruite à la sortie de son bloc d'instructions.

La pile dispose d'un espace limité et l'allocation dynamique d'énormes quantités de mémoire ne se fait pas dans cette zone.

Le tas, une sorte d'arbre (The heap)

Il s'agit de mémoire externe au programme, une demande de mémoire supplémentaire au système d'exploitation. C'est la mémoire utilisée pour l'allocation dynamique. Cinq usages classiques :

- Utilisation d'énormes quantités de mémoire qui déborderait de la zone mémoire du programme.
- Utilisation d'une quantité de mémoire inconnue au moment de la programmation (en c) qui dépend d'une variable comme l'entrée des données.
- Utilisation de structures de données où les blocs ne portent pas de nom comme les variables régulières. Les listes et les arbres peuvent être réordonnés sans copier de données, juste en modifiant les pointeurs.
- Utilisation de structures de données irrégulières comme un tableau de chaines de caractères de différentes longueur.
- Retourner au programme de l'espace alloué à l'intérieur d'une fonction. Lorsque la fonction retourne, la mémoire allouée avec malloc() ou new n'est pas détruite comme les variables déclarées à l'intérieur de celle-ci. Il s'agit donc de retourner via un paramètre ou via la valeur de retour l'adresse de cette mémoire allouée.

Transfert de l'adresse des données : par exemple de la fonction vers le main( )

Puisque l'allocation dynamique requiert la modification du pointeur en lui assignant l'adresse retournée par **new** ou **malloc(),** une fonction accomplissant cette tâche ne peut recevoir une copie du pointeur, le modifier et ne pas le retourner. Toute la mémoire allouée serait perdue Elle doit agir sur le pointeur original (via une référence ou une adresse) ou encore le retourner par la valeur de retour. Trois solutions s'offrent donc à nous :

1. Pour le C++ (passage par référence d'une variable)

On peut envoyer directement le pointeur

```
int main(void)
{
  int **donnees ;
  MonAllocation(donnees, "    fichiersource.txt") ;
  return 0 ;
}
```

Et dire à la fonction de prendre la référence du double pointeur.

```
 void MonAllocation(int ** &t, char nom[])
{
  // l'allocation dynamique
  return;
}
```

Les autres solutions consistesnt à envoyer un pointeur au pointeur (requis en langage C car les références n'existent pas) .

2. Pour le C (envoi de l'adresse du pointeur)

On peut envoyer l'adresse du pointeur

```
int main(void)
{
  int **donnees ;
  MonAllocation(&donnees, "    fichiersource.txt") ;
  return 0 ;
}
```

Et dire à la fonction de référencer le pointeur une fois de plus

```
 Void MonAllocation(int ** *t, char nom[])
{
  // l'allocation dynamique utilise *t
  return;
}
```

ou encore, s'il n'y a qu'un seul pointeur à retourner, utiliser la valeur de retour, tel que font new et malloc().

3. Valeur de retour

On peut ne rien envoyer, juste recevoir la valeur de retour avec l'opérateur = dans le main.

```
int main(void)
{
  int **donnees = MonAllocation("    fichiersource.txt") ;
  return 0 ;
}
```

Et dire à la fonction de retourner le pointeur.

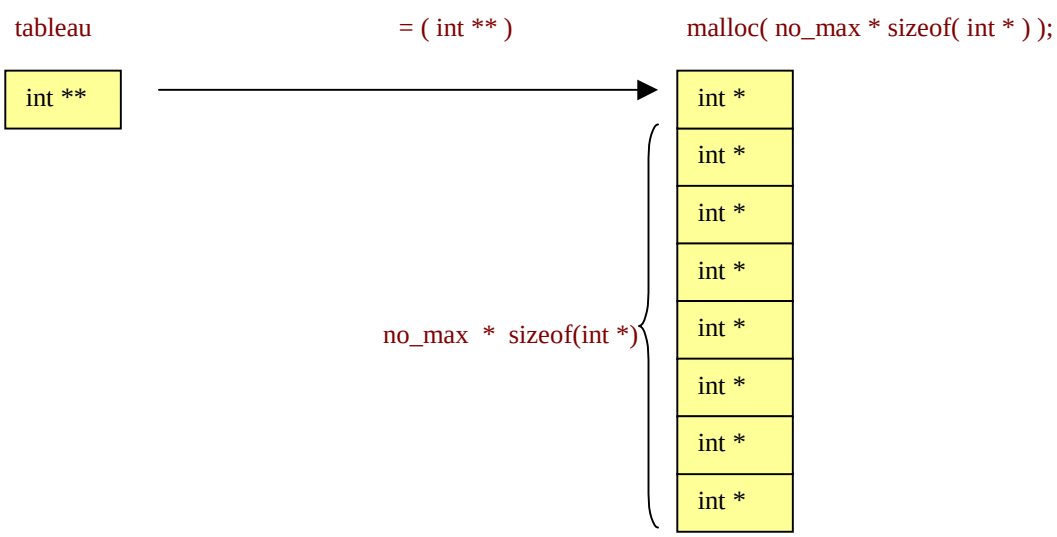
```
int **MonAllocation(char nom[])
{
  int **t ;
  // l'allocation dynamique
  return t;
}
```

4. Oui 4 ! Je le dis tout bas pour ne pas que les autres professeurs entendent… ils ne sont pas d'accord. Si toutes les fonctions de notre programme utilisent cette adresse des données, pourquoi ne pas la déclarer globale (à l'extérieur du main()). Cela économisera bien des copies du pointeur.

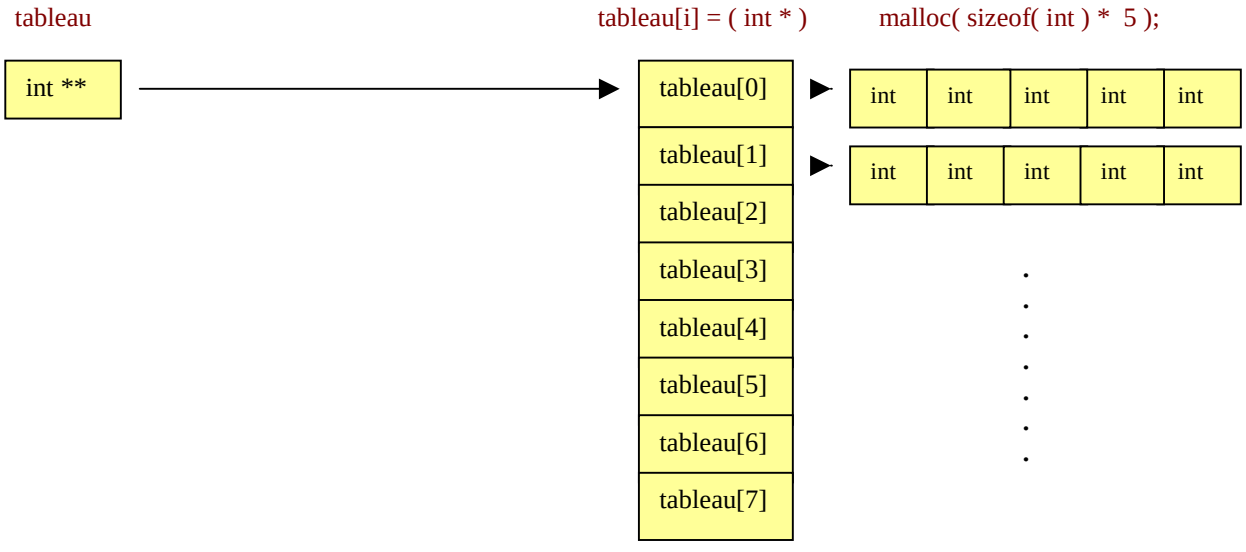
Allocation dynamique en c

```
#include <stdlib.h>
int **tableau;
```


```
tableau = ( int ** ) malloc( sizeof( int * ) * no_max );
```

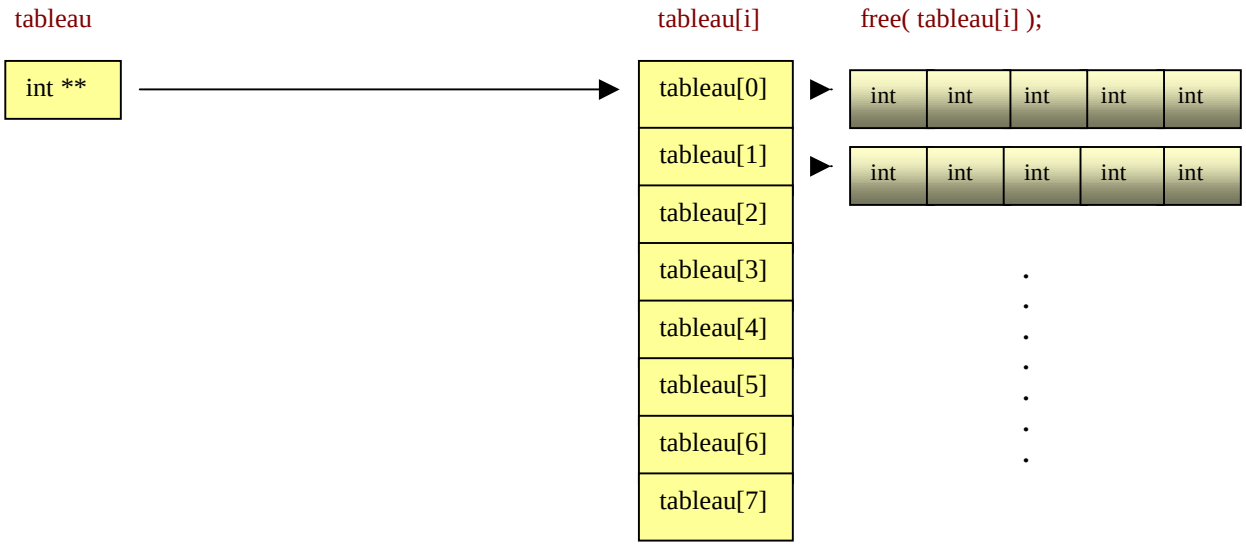



```
for(int i=0; i<no_max; i++)
   tableau[i] = ( int * ) malloc( sizeof( int ) *  5 );
```

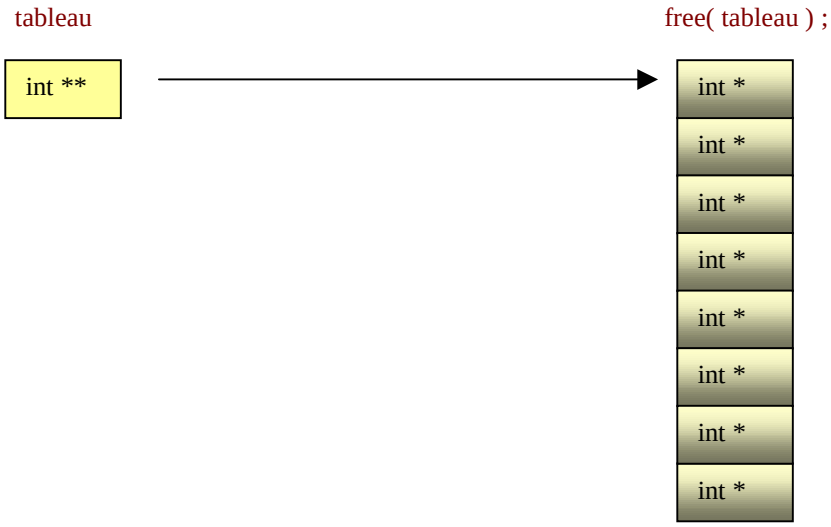


Déallocation en c

```
#include <stdlib.h>
for(int i=0; i<no_max; i++)
   free( tableau[i] );
```



free(tableau);



À la fin il ne reste que le pointeur initial qui avait été déclaré normalement.

Gérer les erreurs d'allocation de mémoire en c

La valeur de retour de malloc est, soit l'adresse de la mémoire allouée en cas de succès, soit zéro en cas d'échec. Il est est important de tester la réussite de l'allocation dynamique pour éviter ultérieurement la référence d'un pointeur null. Dans l'hypothèse qu'une allocation ratée ne sera probablement pas suivie d'une allocation réussie, nous terminons la fonction illico quand un échec survient.

```
#include <stdlib.h>
int **tableau;
tableau = ( int ** ) malloc( no_max *  sizeof( int * ) );
if(tableau == 0)
  return (int **)0;
for(int i=0; i<no_max; i++)
{
  tableau[i] = ( int * ) malloc( 5 * sizeof( int ) );
  if (tableau[i] == 0)
  {
    for(int j=0; j<i; j++ )
      free( tableau[ j ] );
    free( tableau );
    return (int **)0;
  }
}
return tableau ;
```

```
#include <stdlib.h>
int **tableau;
if( ( tableau = ( int ** ) malloc( no_max *  sizeof( int * ) ) ) == 0)
  return (int **)0;
for(int i=0; i<no_max; i++)
{
  if( ( tableau[i] = ( int * ) malloc( 5 *  sizeof( int ) ) ) == 0)
  {
    for(int j=0; j<i; j++ )
      free( tableau[ j ] );
    free( tableau );
    return (int **)0;
  }
}
return tableau ;
```

Fréquentes erreurs de programmation

La mémoire allouée par malloc() devrait être libérée par free() pour usage par le système d'exploitation aussitôt que l'on en n'a plus besoin. Le défaut de faire cela produira un programme gourmand en mémoire vive. L'appel de la fonction free() est relié à l'appel précédent de malloc pour ce même pointeur : un free() pour un malloc(). La fonction free(pointeur) déalloue la mémoire pointée si cette mémoire provient d'un malloc().

Il est dangereux de faire un free() sur un pointeur qui ne pointe pas à de la mémoire malloc'ée. Par exemple, il ne faut pas faire free() sur un pointeur pointant sur une variable locale automatique. Il ne faut évidemment pas faire free() sur un pointeur s'il a déjà été free() (pas deux fois de suite, pas deux fois sans que malloc intervienne entre les deux…). Une dernière erreur, la plus fréquente, est la tentative d'accès à de la mémoire libérée. Une fois que la mémoire est déallouée par free(), il est illégal de référencer le pointeur avec *pointeur ou pointeur[4].