Allocation dynamique et mémoire

Driss MATROUF
MCF HDR

La pile et le tas

- La mémoire est une succession d'octets (8 bits), organisés les uns à la suite des autres et directement accessibles par une adresse.
- En C/C++, la mémoire pour stocker des variables est organisée en deux catégories :
  - La pile (stack)
  - Le tas (heap)
- Attention, d'autres zones mémoire permettant de ranger d'autres informations telles que les programmes eux même, les variables globales, existent.
- Dans les langages de programmation compilés, la pile est l'endroit où sont stockés les paramètres d'appel et les variables locales des fonctions

```
bool distance(Point A, Point B)
{
        double r ;
        ....
}
```

Dans ce cas A, B et r sont stockés dans la pile

La pile (Stack)

- La pile est un espace mémoire réservé au stockage des variables désallouées automatiquement.

```
....
{
        Point A ;
        ...
} // ici disparition de A (libération de A)(désallocation de A)
```

- Sa taille est limitée mais on peut la régler (ulimit -a sous ubuntu)

```
{
        int T[1000000] ;
        ......
}
```

à éviter !!!!

```
matrouf@DESKTOP-H6U0D74:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7823
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7823
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
matrouf@DESKTOP-H6U0D74:~$
```

- La pile utilise la politique LIFO (last in first out) ce qui signifie « dernier entré premier sorti ».
  - Lorsqu'un objet est dépilé il est effacé de la mémoire

Le tas (heap)

- Le tas est l'autre segment de la mémoire utilisé lors de l'allocation dynamique de la mémoire durant l'exécution du programme.
- Sa taille est souvent considérée comme illimitée mais elle est en réalité limitée
  - Attention au swap
- Les opérateurs du langage C++ new et delete permettent respectivement d'allouer et désallouer la mémoire sur le tas.
- Contrairement à la mémoire dans la pile, la mémoire allouée dans le tas doit être désallouée explicitement.
- La taille de la pile limitée (ici 8Mo), Il est évidemment recommandé d'allouer dans le tas les « grosses » variables sous peine de surprises !

Adresse d'une variable : un nouveau type

- Toute variable est accessible grâce à son adresse : son emplacement en mémoire, c'est un entier indiquant le numéro d'octet ou commence la variable.
- L'opérateur & appliqué à une variable donne une valeur : l'adresse de la variable

```
{
    int a ;
    cout << & a<<endl ;
} // affiche : 0x7ffff5a3b524
```

- 0x7ffff5a3b524 en entier vaut « 140736805945860 » est l'adresse de la variable *a*
- &a est une valeur de type adresse d'un entier, on dit aussi de type pointeur sur un entier.
- Le nouveau type est noté : int *, car la variable est de type entier
- Pour une variable a de type float, le type adresse (pointeur) est noté : float *,
-

```
{
  Point A ;
  cout<< & A<<endl ;
}
```

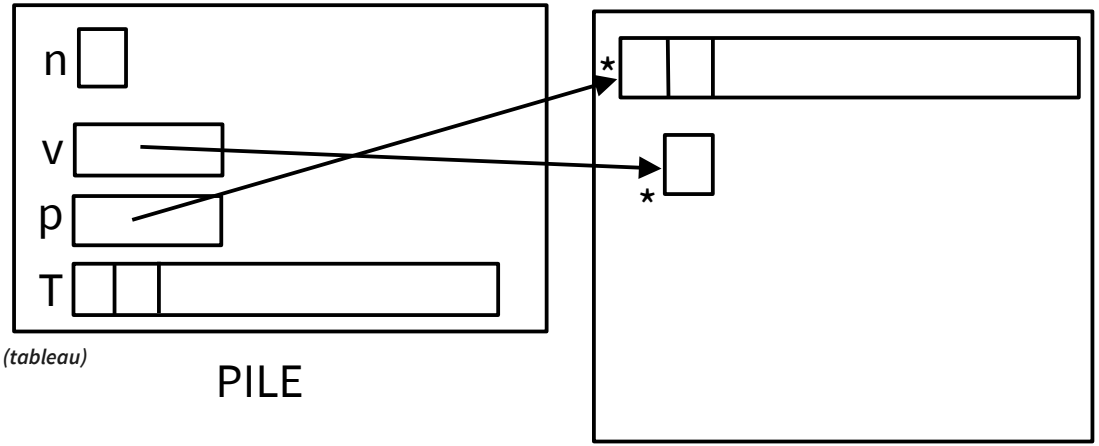
  &A est de type Point * ;
- Ici & intervient en que opérateur (appliqué à une variable) et * intervient dans une assignation de type

New et Delete

- L'allocation mémoire avec new se fait dans un le tas
- *{*

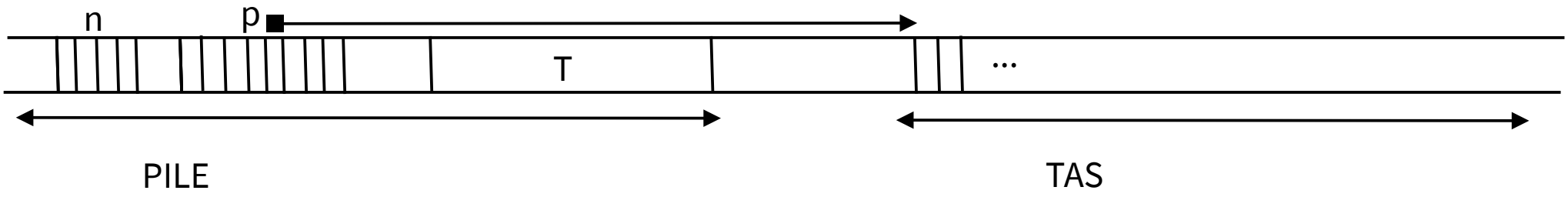
```
int *p ; // p est une variable de type adresse (pointeur)
int n ;
cout<<"Entrez le nombre d'entiers que vous voulez allouer :" ;
cin >> n ;
p=new int[n] ;
….
….
int *v ;
v=new int ;
…
int T[100] ;
delete [] p ; // avec des crochets car il s'agit de l'allocation d'un ensemble de variables (tableau)
delete v ; // pas des crochets car il s'agit de l'allocation d'une seule variable
```

}



- p est une variable de type adresse sur entier
- Je peux allouer un emplacement (dans le tas) pour n entier : p=new int[n] ; new renvoie une adresse où est réservé l'emplacement mémoire
- p est maintenant un tableau d'entiers
- Je peux aussi allouer un seul entier : v=new int ;
- A la fin du bloc : l'espace alloué (dans la pile) avec « int T[100] » est restitué au système, contrairement à celui alloué dans le tas avec « p=new int[n] »
- Pour désallouer l'espace mémoire réservé avec « *p=new int[n] ;* » *il faut le faire explicitement avec « delete [] p »*
- *Pour désallouer* l'espace mémoire réservé avec « v=*new int ;* » *il faut le faire explicitement avec « delete v »*

La mémoire est unidimensionnelle



Allocation, structure

- #include <iostream>

```
struct Point
{
        int x ;
        int y ;
} ;
int main()
{
        Point *p ; // un pointeur sur une variable de type Point
        p=new Point ; // j'alloue dynamiquement une variable de type Point
        cout<< "("<<p→x<<","<<p→y<<")"<<endl ;// Comme p est une adresse (un pointeur) j'utilise → à la place de « . »
        delete p ;
        Point *t ; // un pointeur sur une variable de type Point
        int n=300 ;
        t=new Point[n] ; // j'alloue dynamiquement (dans le tas) un espace mémoire pour contenir 300 Points)
        t[62].x=3 ; t[62].y=3 ; // t est un tableau de points
}
```

Les rôles de * et de &

- **Dans une déclaration (dans une assignation de type)**
  - int *p ; // p est un pointeur vers un entier
  - Point *p ;// p est est un pointeur vers un point
  - int & a ; // a est une variable de type entier passée ou récupérée par référence (ou par variable)
  - Point & a ; // a est une variable de type Point passée ou récupérée par référence
- **En tant qu'opérateur**
  - L'opérateur & s 'applique à une variable
    - *int x=5 ; cout<<(long) &x<<endl ; // & retourne l'adresse d'une variable (c'est une valeur de type pointeur)*
    - *int *p=&x ; // * intervient ici pour assigner le type pointeur à la variable p : p et &x ont le même type*
    - *Point P ; cout<<(long) &P<<endl ; // & retourne l'adresse d'une variable (c'est une valeur de type pointeur)*
    - *Point *a=&P ; // * intervient ici pour assigner le type pointeur à la variable a : a et &P ont le même type*
  - L'opérateur * s'applique à une adresse (un pointeur)
    - *cout<< *p<<endl ; //affiche 5, la valeur contenue dans la variable pointés par p c'est-à-dire x*
    - *a c'est P : *a c'est l'objet pointé par a, c'est-à-dire P

Exemple1

Ecrire une fonction qui :

- Demande à l'utilisateur une valeur entière n

(correspondant à la taille du tableau qu'il veut construire), cette valeur doit être renvoyée à par la fonction

- Elle doit créer dynamiquement un tableau de n entiers

- Elle doit remplir le tableau de valeurs aléatoires entre 0 et 100

```
int * Creer_et_remplir(int & n)
{
    cout<< "Entrez n :" ;
    cin>>n ;
    int *r=new int[n] ;
    for(int i=0;i<n;i++) r[i]=rand()%101 ;
    return (r) ;
}

void afficher(int *T, int n)
{
    for(int i=0;i<n;i++) cout<<T[i]<<" " ;
    cout<<endl ;
}
int main()
{
    // Créer la variable taille pour la donner à la fonction Creer_et_remplir
    int taille;
    // Créer la variable T de type pointeur pour récupérer l'adresse renvoyé par la fonction Creer_et_remplir
    int *T=Creer_et_remplir(taille) ;
    afficher(T,taille) ;
    // à partir de maintenant T est un tableau d'entiers
    ……
}
```

Exemple2

Ecrire une fonction qui prend en argument un tableau de Points et sa taille et qui renvoie un tableau de Points crée dynamiquement sur mesure contenant tous les points dont l'abscisse est supérieure à l'ordonnée. N'oubliez pas qu'il faut retourner la taille du nouveau tableau.

```
Point * Selection_Points(Point *T, int N, int &taille_selection)
{
   taille_selection=0 ;
   for(int i=0;i<N;i++) if(T[i].x >= T[i].y) taille_selection=taille_selection+1 ;
   if(taille_selection>0)
   {
      Point *R=new Point[taille_selection] ;
      int k=0 ;
      for(int i=0;i<N;i++) if(T[i].x >= T[i].y) {R[k]=T[i] ; k++;}
      return R ;
   }
   else return( NULL) ;
}
```

Exemple 3

f- Écrire une fonction qui prend en argument un tableau de d'entiers et sa taille et qui renvoie 2 tableaux d'entiers crées dynamiquement sur mesure contenant d'une part toutes les valeurs positives et d'autre part toutes les valeurs négative. N'oubliez pas qu'il faut retourner les tailles des nouveaux tableaux. Attention on ne peut pas allouer un tableau de taille 0.

Correction f :

Voici le prototype

```
void Selection_Points(Point *T, int N, int * & T1, int * & T2, int & taille1, int & taille2)
T1 et T2 sont deux pointeurs passés par variable
T1=new int[taille1] ;
T2=new int[taille2] ;
```