Algorithmique/Langage
1ère année

Pointeurs et Allocation Dynamique

Yacine BELLIK

IUT d'Orsay
Université Paris XI

Plan du cours

Pointeurs

Définition

- Un pointeur p est une variable qui sert à contenir l'adresse mémoire d'une autre variable[(1)] v
- Si la variable v occupe plus d'une case mémoire, l'adresse contenue dans le pointeur p est celle de la première case utilisée par v
- La variable v est appelé variable pointée
- Si p contient l'adresse de v alors on dit que p pointe sur v
- Les pointeurs possèdent un type qui dépend du type des variables sur lesquelles ils auront à pointer
  - Exemple
    - un pointeur d'entiers servira à contenir l'adresse de variables entières
    - un pointeur de réels servira à contenir l'adresse de variables réelles
    - etc.

(1) *Un pointeur peut également contenir l'adresse d'une fonction. On parlera alors de pointeurs de fonctions.*

Déclaration de pointeurs

- Lorsqu'on déclare un pointeur on indique le type des variables sur lesquelles il aura à pointer

```
char  * pc; // pc est un pointeur de caractères
int   * pi; // pi est un pointeur d'entiers
float * pf; // pf est un pointeur de flottants
```

- Déclaration de plusieurs pointeurs sur une même ligne

```
int * pi1, i1, * pi2, i2;
```

  - pi1 et pi2 sont des pointeurs d'entiers
  - i1 et i2 sont des entiers
  - il faut répéter * pour chaque pointeur

Adresse d'une variable: opérateur &

- L'adresse d'une variable peut être obtenue grâce à l'utilisation de l'opérateur & suivi du nom de la variable

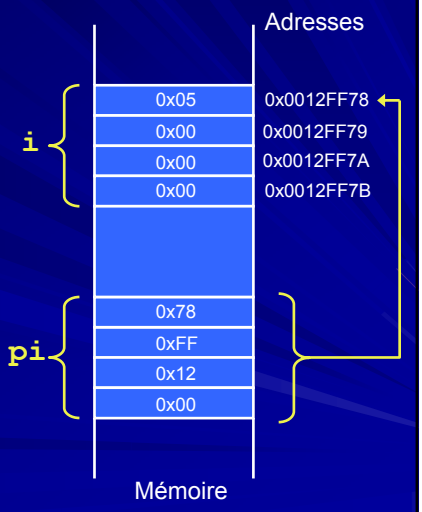
```
int i=5;
cout << "Valeur  de i = " << i  << endl;
cout << "Adresse de i = " << &i << endl;

Trace d'exécution :

Valeur de  i = 5
Adresse de i = 0x0012FF78
```

Affectation d'une adresse à un pointeur

```
int i=5;
int * pi;
pi=&i;
cout << "Valeur  de i = " << i  << endl;
cout << "Adresse de i = " << &i << endl;
cout << "Valeur de pi = " << pi << endl;
```

Trace d'exécution :

```
Valeur de  i = 5
Adresse de i = 0x0012FF78
Valeur de pi = 0x0012FF78
```



Incompatibilité de types

- On ne peut pas affecter à un pointeur de type T1 l'adresse d'une variable de type T2 différent de T1

```
int i=5;
char * pc;
pc=&i; // !!! ERREUR !!!
```

- On peut cependant forcer la conversion.
- A utiliser avec précaution...

```
int i=5;
char * pc;
pc=(char*) &i;
```

Opérateur d'indirection *

- Cet opérateur permet d'accéder à la valeur de la variable pointée en utilisant le pointeur
- Si p contient l'adresse de la variable v alors *p désigne la variable v elle-même

```
int i=5;
int * pi=&i;
cout << "Valeur  de i = " << i   << endl;
cout << "Adresse de i = " << &i  << endl;
cout << "Valeur de pi = " << pi  << endl;
cout << "Valeur  de i = " << *pi << endl;
```

Trace d'exécution :

```
Valeur de  i = 5
Adresse de i = 0x0012FF78
Valeur de pi = 0x0012FF78
Valeur de  i = 5
```


Opérateur d'indirection *

- l'opérateur * permet de lire la valeur de la variable pointée mais également de la modifier

```
int i=5;
int * pi=&i;
cout << "Valeur  de i = " << i << endl;
*pi=9;
cout << "Valeur  de i = " << i << endl;
```

Trace d'exécution :

```
Valeur de  i = 5
Valeur de  i = 9
```

Exercice

```
int v1 = 5, v2 = 15;
int *p1, *p2;

p1 = &v1;        // p1 = adresse de v1
p2 = &v2;        // p2 = adresse de v2
*p1 = 10;        // variable pointée par p1 = 10
*p2 = *p1;       // variable pointée par p2 = variable pointée par p1
p1 = p2;         // p1 = p2
*p1 = 20;        // variable pointée par p1 = 20

cout << "v1==" << v1 << " v2==" << v2;
```

- Que va produire ce programme ???

Devinette

```
int i=5, j=2;
int *pi, *pj;
pi=&i; pj=&j;
cout << *pi**pj << endl;
cout << *pi/*pj << endl;
```

- Que va produire ce programme ???

Pointeurs void

- Les pointeurs void sont un type particulier de pointeur
- Ils peuvent pointer sur une variable de n'importe quel type
- On ne peut pas utiliser l'opérateur d'indirection * sur un pointeur void. Il faut d'abord le convertir en un pointeur d'un type donné.

```
int a=5, b=6;
int  * pi=&a;
void * pv=&b;          // Correct

pv=pi;                 // Correct
pi=pv;                 // !!! Erreur !!!
pi= (int *) pv;        // Correct
cout<< *pv;            // !!! Erreur !!!
cout<< *((int*)pv);    // Correct
```


Pointeurs d'objets

```
Point p(10,20);
Point * pp=&p;

// on peut écrire
cout << "x=" << p.getAbs() << " y=" << p.getOrd() << endl;
// ou encore (parenthèses obligatoires)
cout << "x=" << (*pp).getAbs() << " y=" << (*pp).getOrd() << endl;
// ou encore
cout << "x=" << pp->getAbs() << " y=" << pp->getOrd() << endl;
```

- Dans le cas de pointeurs d'objets, il est plus pratique d'utiliser l'opérateur → que l'opérateur *
- Cette syntaxe s'applique également aux donnés membres

Adresse de l'objet cible

```
void Rectangle::afficher() const
{
   cout << "x1= " << _x1 << endl;
   cout << "y1= " << (*this)._y1 << endl;
   cout << "x2= " << this->_x2 << endl;
   cout << "y2= " << _y2 << endl;
}
```

- this représente l'adresse de l'objet cible
- *this représente donc l'objet cible lui-même

Opérateur sizeof

- Cet opérateur permet de connaître la taille en octets d'une constante, d'une variable ou d'un type

```
const int MAX=100;

char c='A'; int i=10; float f; char * pc=&c;
char t1[]="bonjour"; int t2[MAX];

cout<<"Taille de MAX="<<sizeof(MAX)<<endl;
cout<<"Taille de c="<<sizeof(c)<<" Taille de char="<<sizeof(char)<<endl;
cout<<"Taille de i="<<sizeof(i)<<" Taille de int="<<sizeof(int)<<endl;
cout<<"Taille de f="<<sizeof(f)<<" Taille de float="<<sizeof(float)<<endl;
cout<<"Taille de pc="<<sizeof(pc)<<" Taille de pointeur="<<sizeof(char*)<<endl;
cout<<"Taille de t1="<<sizeof(t1)<<endl;
cout<<"Taille de t2="<<sizeof(t2)<<endl;
```

Opérateur sizeof

- Trace d'exécution

```
Taille de MAX=4
Taille de c=1  Taille de char=1
Taille de i=4  Taille de int=4
Taille de f=4  Taille de float=4
Taille de pc=4 Taille de pointeur=4
Taille de t1=8
Taille de t2=400
```


Incrémentation / Décrémentation d'un pointeur

- Soit p un pointeur de type T
- Soit t la taille en octets du type T
- Soit a la valeur de p (adresse contenue dans p)
- Si on exécute p++ alors la nouvelle valeur de p sera égale à a+t

```
int i=5; int * pi=&i;
cout << "Valeur de pi avant incrémentation = " << pi << endl; pi++;
cout << "Valeur de pi après incrémentation = " << pi << endl;
Trace d'exécution :
Valeur de pi avant incrémentation = 0x0012FF78
Valeur de pi après incrémentation = 0x0012FF7C
```

```
double x=1.25; double * px=&x;
cout << "Valeur de px avant incrémentation = " << px << endl; px++;
cout << "Valeur de px après incrémentation = " << px << endl;
Trace d'exécution :
Valeur de px avant incrémentation = 0x0012FF78
Valeur de px après incrémentation = 0x0012FF80
```

- Le même principe s'applique pour p--

Addition / Soustraction d'un entier

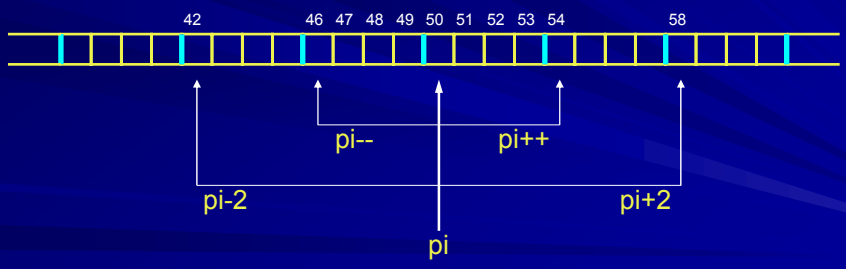
- Soit p un pointeur de type T
- Soit t la taille en octets du type T
- Soit a la valeur de p (adresse contenue dans p)
- Soit n un entier
- Si on exécute p+n alors la nouvelle valeur de p sera égale à a+n*t

```
int i=5;
int * pi=&i;
cout << "Valeur de pi avant l'addition = " << pi << endl;
pi=pi+2;
cout << "Valeur de pi après l'addition = " << pi << endl;

Trace d'exécution :
Valeur de pi avant l'addition = 0x0012FF78
Valeur de pi après l'addition = 0x0012FF80
```

- Le même principe s'applique pour la soustraction d'un entier

Exemple récapitulatif

- Soit pi un pointeur d'entier



Tableaux et pointeurs

- Les identificateurs de tableaux et de pointeurs sont très similaires
- L'identificateur d'un pointeur désigne l'adresse de la première case mémoire de la variable sur laquelle pointe le pointeur
- De même, l'identificateur d'un tableau désigne l'adresse de la première case mémoire du tableau
- Il existe cependant une différence majeure
  - La valeur du pointeur peut être modifiée
  - L'adresse du tableau ne peut pas être modifiée
- L'identificateur d'un tableau peut être considéré comme un **pointeur constant**

Tableaux et pointeurs

```
const int MAX=7;
int tab[MAX]; int * p;
p=tab;      *p=0;        // tab[0]=0
p++;        *p=10;       // tab[1]=10
p=&tab[2];  *p=20;       // tab[2]=20
p=tab+3;    *p=30;       // tab[3]=30
p=tab;      *(p+4)=40;   // tab[4]=40
p[5]=50;                 // tab[5]=50
*(tab+6)=60;             // tab[6]=60;

for (int n=0; n<MAX; n++) cout << tab[n] << " ";

Trace d'exécution :
0 10 20 30 40 50 60
```

- Une instruction telle que **tab=p** aurait provoqué une erreur à la compilation

Affichage de tableaux et de pointeurs

- L'affichage d'un tableau ou d'un pointeur produit des résultats différents selon qu'il s'agisse d'un tableau ou pointeur de caractères ou pas.
- Dans le cas d'un tableau ou pointeur de caractères, l'affichage produit la chaîne de caractères commençant à la première case du tableau ou à la case d'adresse indiquée par le pointeur et finissant à la première case qui contient le caractère \0 (code ASCII=0).
- Dans les autres cas (tableau ou pointeur d'un autre type), l'affichage produira l'adresse de la première case du tableau ou la valeur du pointeur (adresse contenue dans le pointeur).
- Pour obtenir l'affichage de la valeur d'un pointeur de caractères (c'est-à-dire l'adresse qu'il contient) il faut le convertir en pointeur void avant.

Affichage de tableaux ou de pointeurs autres que les tableaux ou pointeurs de caractères

```
const int MAX=10;
int * p;
int t[MAX]={1,2,3};

p=t;
cout << " t=" <<  t << "   p=" <<  p << endl;
cout << "&t=" << &t << "  &p=" << &p << endl;
cout << "*t=" << *t << "  *p=" << *p << endl;
```

Trace d'exécution

```
 t=0x0012FF54      p=0x0012FF54
&t=0x0012FF54     &p=0x0012FF50
*t=1              *p=1
```

Affichage de tableaux ou de pointeurs de caractères

```
const int MAX=10;
char * p;
char t[MAX]="ABCDEF";

p=t;
cout << " t=" <<  t << "   p=" <<  p << endl;
cout << "&t=" << &t << "  &p=" << &p << endl;
cout << "*t=" << *t << "  *p=" << *p << endl;
cout << "(void*) p=" << (void*) p <<endl;
```

Trace d'exécution

```
 t=ABCDEF          p=ABCDEF
&t=0x0012FF54     &p=0x0012FF50
*t=A              *p=A
(void*) p=0x0012FF54
```


Passage de paramètres en C

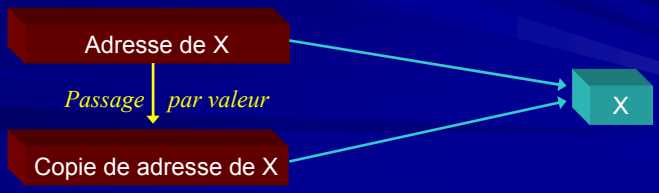
- Passage de paramètres en C++
  - Par valeur
  - Par référence
- Passage de paramètres en C
  - Par valeur uniquement
- Problème
  - Comment passer en C un paramètre en donnée/résultat ?
- Solution
  - Passer son adresse en paramètre

Exemple 1

- Procédure permuter : permute 2 variables

```
// C++
void permuter (int & x, int & y)
{
    int temp=x;
    x=y;
    y=temp;
}
// Appel
permuter (a,b);
```

```
// C
void permuter (int * px, int * py)
{
    int temp=*px;
    *px=*py;
    *py=temp;
}
// Appel
permuter (&a,&b);
```



Exemple 2

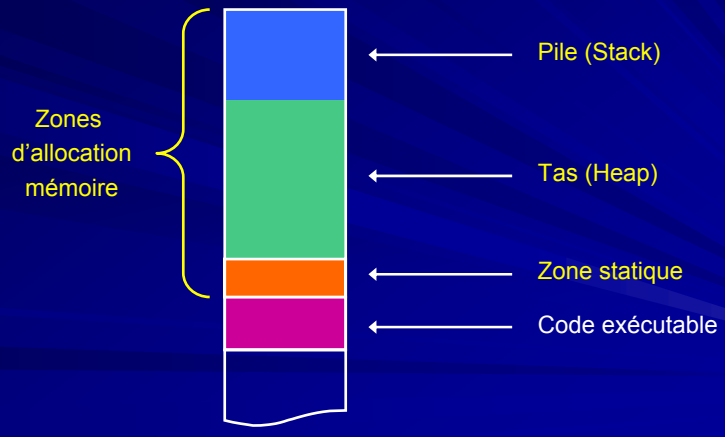
- Procédure minmax : renvoie le min et le max d'un tableau d'entiers

```
// On suppose que le tableau contient au moins 1 élément
void minmax (int tab[], int n, int * pmin, int * pmax)
{
   *pmin=*pmax=tab[0];
   for(int i=1;i<n;i++)
   {
        if ( tab[i] < *pmin ) *pmin=tab[i];
        else if ( tab[i] > *pmax ) *pmax=tab[i];
   }
}
// Appel
minmax(t,100,&min,&max);
```

Allocation dynamique

Structure de la mémoire

- La mémoire comporte 3 zones d'allocation différentes
- Chaque zone sert à mémoriser 3 types différents de variables



La pile

- La pile contient les variables qui sont déclarées à l'intérieur des fonctions ou à l'intérieur des blocs ({…})
- Ces variables sont appelées variables locales ou automatiques
- Elles sont créés à l'appel de la fonction et détruites à la sortie de la fonction (respectivement à l'entrée et à la sortie du bloc)
- Leurs durées de vie est donc la durée de vie de la fonction ou du bloc

La zone statique

- La zone d'allocation statique contient les variables qui sont déclarées en dehors de toute fonction ou à l'intérieur d'une fonction mais avec le qualificatif static
- Ces variables sont appelées variables globales ou statiques
- Elles sont créés à l'exécution du programme et détruites à la fin de celui-ci
- Leurs durées de vie est donc la durée de vie du programme

Le tas

- Le tas contient les variables qui sont créés par le programme au cours de son exécution. On parle alors d'**allocation dynamique**.
- Ces variables sont appelées variables dynamiques
- Elles sont créés lorsque le programme utilise l'opérateur **new** et détruites lorsqu'il utilise l'opérateur **delete**
- Leurs durées de vie est donc variable (commence à l'exécution du **new** et se termine à l'exécution du **delete**)
- Si le programmeur oublie d'appeler **delete** (ce qui est un signe de mauvaise programmation!) sur une variable créée avec **new**, celle-ci sera quand même détruite à la fin de l'exécution du programme.

Intérêt de l'allocation dynamique

- L'allocation dynamique permet une gestion plus flexible de la mémoire.
- Un programme n'utilise la mémoire dont il a besoin qu'au moment où il en a besoin.
- Ainsi il évite de monopoliser de l'espace mémoire aux instants où il n'en a pas besoin et permet donc à d'autres programmes de l'utiliser.
- La mémoire peut ainsi être partagée de manière plus efficace entre plusieurs programmes.
- Il est impératif d'appeler l'opérateur **delete** lorsqu'on n'a plus besoin de l'espace mémoire correspondant sinon l'allocation dynamique perd tout son intérêt.

Intérêt de l'allocation dynamique

- L'allocation dynamique sert les autres programmes mais elle peut également servir le programme même qui l'utilise.
- Supposons qu'un programme ait besoin d'afficher 100 objets graphiques différents et que ces objets sont très volumineux (en espace mémoire).
- Supposons également qu'un instant donnée, le programme n'a besoin d'afficher simultanément et au maximum que 10 objets parmi les 100.
- Supposons que la taille de la mémoire permet au maximum le stockage simultané de 30 objets graphiques.
- Allouer les 100 objets de façon statique est impossible puisque l'espace mémoire est insuffisant.
- En revanche, déclarer dynamiquement 10 objets est tout à fait possible et le programme pourra alors s'exécuter sans problèmes.

Opérateur new

- L'opérateur new permet l'allocation dynamique de l'espace mémoire nécessaire pour stocker un élément d'un type T donné
- Pour que l'opérateur new puisse connaître la taille de l'espace à allouer il faut donc lui indiquer le type T
- Si l'allocation réussit (il y a suffisamment d'espace en mémoire) alors l'opérateur new retourne l'adresse de l'espace alloué
- Cette adresse doit être alors affectée à un pointeur de type T pour pouvoir utiliser cet espace par la suite
- Si l'allocation échoue (il n'y a pas suffisamment d'espace en mémoire) alors l'opérateur new retourne NULL (0)

```
T * p;
p = new T;
```

ou

```
T * p = new T;
```

- Il est possible d'indiquer après le type T une valeur pour initialiser l'espace ainsi alloué. Cette valeur doit être indiquée entre parenthèses : new T (val)

Opérateur new : exemple

- Allocation d'un entier

```
//Sans initialisation
int * p = new int;
*p=5;
cout << *p;  // 5
```

```
//Avec initialisation
int * p = new int (10);
cout << *p;  // 10
```

- Allocation d'un objet

```
int * p1 = new Point; //Allocation d'un objet Point et appel du constructeur sans paramètres
                      (ou avec les valeurs par défaut)
int * p2 = new Point(30,40); //Allocation et appel du constructeur avec paramètres

cout << p1->getAbs() << " " << p1->getOrd() << endl;  // 0 0
cout << p2->getAbs() << " " << p2->getOrd() << endl;  // 30 40
```

- Nous n'avons pas vérifié ici si l'allocation a réussi. Dans la pratique il faudra le faire systématiquement.

Opérateur new [ ]
Allocation dynamique d'un tableau 1D

- Pour allouer dynamiquement un tableau d'éléments il suffit d'utiliser l'opérateur **new [ ]** en indiquant entre les crochets le nombre d'éléments désiré
- Contrairement aux tableaux statiques où le nombre d'éléments devait être une constante (valeur connue à la compilation) ici le nombre d'éléments peut être une variable dont la valeur ne sera connue qu'à l'exécution (valeur saisie par l'utilisateur, lue à partir d'un fichier, calculée,…)
- On récupère alors l'adresse du premier élément du tableau

```
T * p = new T [n]; // p contiendra l'adresse de T[0]
```

- Il n'est pas possible ici d'indiquer des valeurs d'initialisation
- Dans le cas d'un tableau d'objets il faut obligatoirement prévoir un constructeur sans paramètres ou avec des valeurs pas défaut

Opérateurs delete et delete [ ]

- Pour que l'allocation dynamique soit utile et efficace, il est impératif de libérer l'espace réservé avec new dès que le programme n'en a plus besoin.
- Cet espace pourra alors être réutilisé soit par le programme lui-même soit par d'autres programmes.
- Libération d'un élément simple : **opérateur delete**

```
int * p = new int;
*p=5;
cout<<*p<<endl;
delete p;
```

- Libération d'un tableau : **opérateur delete [ ]**

```
int * p, i, n;
cout<<"Nombre d'éléments ? : "; cin>>n;
p=new int[n];
for (i=0;i<n;i++) cin>>p[i];
for (i=0;i<n;i++) cout<<p[i]*p[i];
delete [] p;
```


Gestion des erreurs d'allocation

- On peut gérer gérer les erreurs d'allocation dynamique de la mémoire de 3 façons différentes
  - Test de la valeur retournée par new après chaque allocation
  - Utilisation de set_new_handler
  - Utilisation de l'exception standard bad_alloc

Test de la valeur retournée par new

```
const int CENTMEGAS=100*1024*1024;
char * p;
int i=0;
while(1)
{
    p=new (nothrow) char[CENTMEGAS];
    if (p) cout<<++i<<"00 Mégas octets alloués"<<endl;
    else  {cerr<<"Plus de mémoire"<<endl; exit(1);}
}
```

```
100 Mégas octets alloués
200 Mégas octets alloués
300 Mégas octets alloués
400 Mégas octets alloués
500 Mégas octets alloués
600 Mégas octets alloués
700 Mégas octets alloués
800 Mégas octets alloués
900 Mégas octets alloués
1000 Mégas octets alloués
1100 Mégas octets alloués
1200 Mégas octets alloués
1300 Mégas octets alloués
1400 Mégas octets alloués
1500 Mégas octets alloués
Plus de mémoire
```

- Lourde car nécessite un test après chaque new
- nothrow sert à demander à new de ne pas lever d'exception (sinon le test qui suit ne sert à rien...)

set_new_handler

- On indique une fonction (dans l'exemple erreurMemoire) qui sera appelée automatiquement si new échoue
- Inclure <new>

```
void erreurMemoire() {
    cerr<<"Plus de mémoire"<<endl;
    exit(1);
}
int main() {
    const int CENTMEGAS=100*1024*1024;
    char * p; int i=0;

    set_new_handler(&erreurMemoire);
    while(1){
       p=new char[CENTMEGAS];
       cout<<++i<<"00 Mégas octets alloués"<<endl;
    }
    return 0;
}
```

```
100 Mégas octets alloués
200 Mégas octets alloués
300 Mégas octets alloués
400 Mégas octets alloués
500 Mégas octets alloués
600 Mégas octets alloués
700 Mégas octets alloués
800 Mégas octets alloués
900 Mégas octets alloués
1000 Mégas octets alloués
1100 Mégas octets alloués
1200 Mégas octets alloués
1300 Mégas octets alloués
1400 Mégas octets alloués
1500 Mégas octets alloués
Plus de mémoire
```

Exception standard bad_alloc

- L'exception standard bad_alloc est levée automatiquement si new échoue
- Inclure <stdexcept>

```
const int CENTMEGAS=100*1024*1024;
char * p; int i=0;

try
{
    while(1)
    {
          p=new char[CENTMEGAS];
          cout<<++i<<"00 Mégas octets alloués"<<endl;
    }
}
catch(bad_alloc & e)
{
    cerr<<"Plus de mémoire"<<endl;
    cerr<<e.what()<<endl;
    terminate();
}
```

```
100 Mégas octets alloués
200 Mégas octets alloués
300 Mégas octets alloués
400 Mégas octets alloués
500 Mégas octets alloués
600 Mégas octets alloués
700 Mégas octets alloués
800 Mégas octets alloués
900 Mégas octets alloués
1000 Mégas octets alloués
1100 Mégas octets alloués
1200 Mégas octets alloués
1300 Mégas octets alloués
1400 Mégas octets alloués
1500 Mégas octets alloués
Plus de mémoire
St9bad_alloc
Aborted (core dumped)
```


Tableaux dynamiques à 2 dimensions

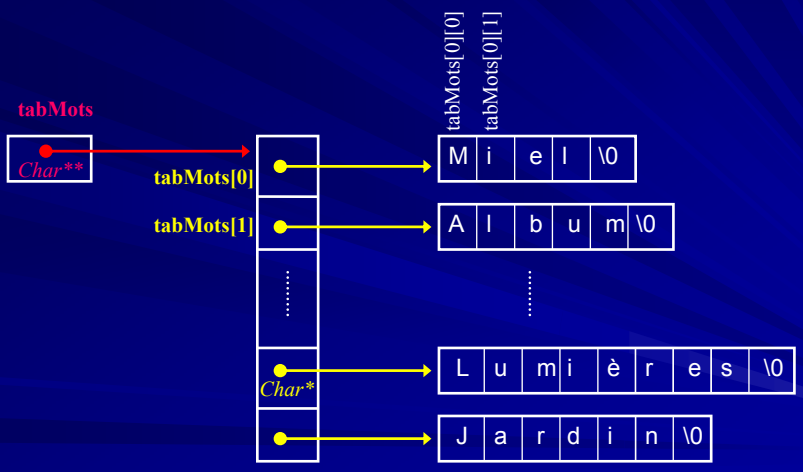
- Il n'est pas possible en C++ d'allouer dynamiquement un tableau en 2 dimensions de la même façon que cela se fait pour les tableaux statiques
- La valeur de la deuxième dimension doit être connue à la compilation

```
new int [i][10]; // OK
```

```
new int [i][j]; // Erreur !!!
```

- Il existe cependant différentes solutions pour simuler un tableau à 2 dimensions (ou à n dimensions)
- Le choix de la meilleure solution dépend de différents paramètres
  - L'usage qui sera fait du tableau
    - Accès séquentiel, par ligne, par colonne,…
    - Accès aléatoire
  - Nombre de lignes et/ou colonnes
    - Fixe
    - Variable
  - Avec ou sans gestion des indices
  - …

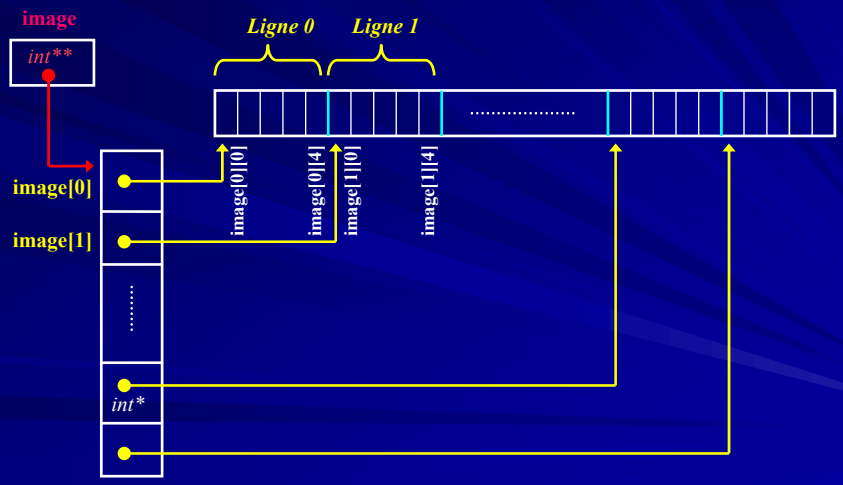
Exemple 1 : tableau de chaînes



Exemple 1 : tableau de chaînes

```
int i, nbMots;
char mot[LONGMAX];
char ** tabMots;

cout<<"nombre de mots ? : "; cin>>nbMots;
tabMots=new char * [nbMots]; //Allocation du tableau de pointeurs

for (i=0;i<nbMots;i++) {
   cout<<"Mot "<<i+1<<" ? : ";
   cin.getline(mot,LONGMAX); //Saisie du ième mot dans une variable temporaire
   tabMots[i]=new char[strlen(mot)+1]; //Allocation de l'espace nécessaire au ième mot
   strcpy(tabMots[i],mot); //Copie du ième mot saisi dans l'espace alloué
}
for (i=0;i<nbMots;i++)cout<<tabMots[i]<<endl;

for (i=0;i<nbMots;i++) delete [] tabMots[i]; // Libération de l'espace de chaque mot
delete [] tabMots; // Libération du tableau de pointeurs
```

Exemple 2 : tableau de pixels (image)



Exemple 2 : tableau de pixels (image)

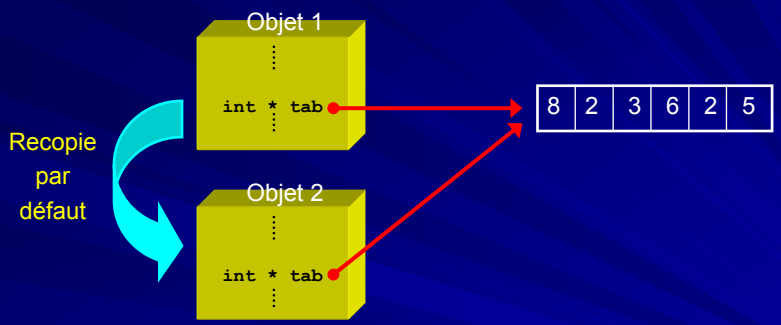
```
int nbCol, nbLig, nbPix, i, j;
Int * tab;
int **image;
cout<<"Nombre de lignes et de colonnes ? : "; cin>>nbLig>>nbCol;
nbPix=nbLig*nbCol;

image=new int* [nbLig]; // Allocation du tableau de pointeurs de lignes
tab=new int[nbPix];     // Allocation du tableau 1D servant à stocker l'image elle-même
for(i=0;i<nbLig;i++) image[i]=tab+i*nbCol; // Remplissage du tableau de pointeurs de lignes

for(i=0;i<nbLig;i++) for(j=0;j<nbCol;j++) cin>>image[i][j]; // lecture de la couleur des pixels
for(i=0;i<nbPix;i++) tab[i]+=10; // augmentation de la saturation de la couleur des pixels
                                 // accès plus rapide que image[i][j]
delete []tab;    // Libération du tableau 1D servant à stocker l'image
delete []image; // Libération du tableau de pointeurs de lignes
```

- Cette solution permet
  - L'utilisation de 2 indices pour un accès aisé
  - Ou l'utilisation d'un seul indice pour un accès plus rapide

Objets ayant une partie dynamique

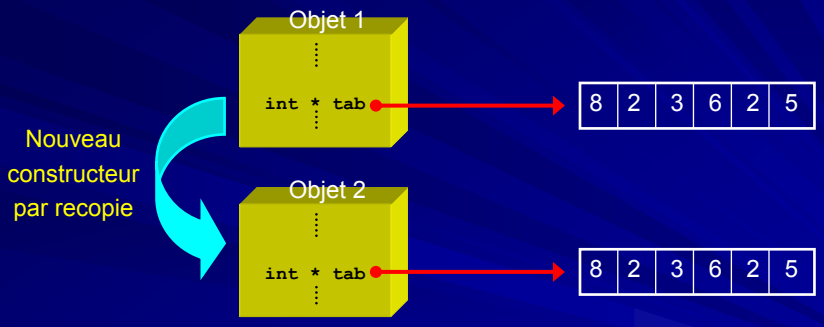
- On entend par objets ayant une partie dynamique, des objets dont le constructeur réalise une allocation dynamique
- Un certain nombre de dispositions doivent être prises avec de tels objets
  - Ecriture d'un destructeur
  - Redéfinition du constructeur par recopie
  - Redéfinition de l'opérateur d'affectation
- Ecriture d'un destructeur
  - Le destructeur doit libérer l'espace alloué par le constructeur
- Redéfinition du constructeur par recopie et de l'opérateur d'affectation
  - Pour éviter dans certains cas le problème de la recopie superficielle

Recopie superficielle



- Le constructeur par recopie par défaut ne faut que recopier les données membres sans s'occuper de ce vers quoi elles pointent
- Les 2 objets deviennent dépendants. Ils partagent le même tableau. Ceci peut parfois conduire à des résultats indésirables.
- Il faut dans ce cas redéfinir le constructeur par recopie et l'opérateur d'affectation de manière à réaliser une copie profonde.

Recopie profonde



- Le nouveau constructeur par recopie doit allouer un nouvel espace et y recopier les valeurs du tableau de l'objet 1
- L'opérateur d'affectation doit faire en plus certaines actions du fait que l'objet existe déjà et possède déjà un espace alloué (voir →)

Opérateur d'affectation

- Le nouvel opérateur d'affectation doit :
  - Faire le test de l'auto-affectation (A=A) et si ce test est vrai se contenter simplement de retourner l'objet cible (par référence)
  - Si le test de l'auto-affectation est faux (les 2 objets sont différents) alors il doit :
    - Libérer le tableau dynamique alloué par l'objet cible
    - Allouer un nouveau tableau à l'objet cible
    - Recopier les valeurs du tableau de l'objet de droite dans le nouveau tableau de l'objet cible
  - Le test de l'auto-affectation se fait en comparant l'adresse de l'objet cible avec celle de l'objet passé en paramètre à l'opérateur d'affectation