Les classes proposent un comportement par défaut extrêmement précieux. C'est ce que nous avons appelé « forme canonique ». Rappelons de quoi il s'agit. Chaque classe est systématiquement composée (implicitement) :

```cpp
//----------------------------------------------------------------
class T { }; // lorsque nous créons cette classe vide
//----------------------------------------------------------------
// nous avons en fait
class T
{
public:
  T();                      // constructeur par défaut
  T(const T&);              // constructeur de copie
  ~T();                     // destructeur
  T& operator= (const T&); // opérateur d'affectation
};
//----------------------------------------------------------------
int main()
{
  T t1;        // t1 objet de T - construction par défaut
  T t2 = t1;   // t2 est construit à partir de t1 (copie)
  t1 = t2;     // utilisation de l'opérateur d'affectation
}              // appel des destructeurs à ce niveau
//----------------------------------------------------------------
```

- ***D'un constructeur par défaut*** : qui ne fait rien par défaut. Ce constructeur est rarement utilisé puisque nous avons souvent besoin de définir un ou plusieurs constructeurs adaptés à chacune des situations, ce qui fait que celui qui est proposé par défaut est généralement occulté par les autres.
- ***D'un destructeur par défaut*** : qui ne fait rien par défaut. Ce destructeur est à redéfinir dans le cas uniquement où il existe au moins une variable dynamique au sein de l'objet, auquel cas, il sera nécessaire de libérer la mémoire de cette variable au moment de la destruction.
- ***D'un constructeur de copie*** : qui propose par défaut une copie membre à membre. Les attributs de l'objet à créer sont initialisés par rapport aux attributs de l'objet (qui sert de copie) passé en argument.
- ***D'un opérateur d'affectation*** : qui propose par défaut également une copie membre à membre.

Nous avons largement traités les deux premiers cas dans nos précédents cours, et nous savons bien comment réagir face à ces diverses situations. Quant aux deux derniers cas, le comportement proposé par défaut parait très séduisant puisque sans écriture particulière, la copie entre objets de même nature est possible. Le tout est de savoir si cette copie proposée par défaut répond à toutes les situations envisagées.

Bien que dans 80% des cas, tout se passe effectivement sans problème, il existe des situations où la copie membre à membre ne fonctionne pas correctement, et où il sera nécessaire de redéfinir le comportement par défaut, ou éventuellement, d'empêcher la copie. Avant d'arriver à ces différentes alternatives, il faudrait d'abord mettre en évidence le problème.

Classe « *TabInt* » - construction et destruction

Nous allons pour cela créer une classe qui représente un tableau d'entier. Cette classe devra remplacer les tableaux classiques. En effet, ces derniers ne permettent pas la copie directe par simple affectation puisque, souvenez-vous, un tableau classique est simplement un pointeur constant. Nous allons donc compléter notre classe successivement pour permettre une utilisation très simple et dans ce cas, il sera nécessaire, au moins, de redéfinir l'opérateur crochet « [ ] » (appelé *opérateur d'indexation*) pour retrouver la même utilisation qu'un tableau classique.

N'anticipons pas, il faut d'abord savoir créer cet objet, et aussi le détruire. En effet, dans la conception d'une classe, il faut généralement s'occuper d'abord de ces deux problèmes avant d'envisager d'autres comportements. Au moment de la création de l'objet, nous devons préciser la dimension (taille) du tableau. A titre d'exemple, vous avez ci-contre une utilisation possible avec un objet « *tab1* » qui correspond à un tableau d'entier de *5* cases.

```cpp
22 int main()
23 {
24   TabInt tab1(5);
25   return 0;
26 }
```

```cpp
1 class TabInt
2 {
3   int *tableau;
4   int taille;
5 public:
6   TabInt(int taille);
7   ~TabInt();
8 };
```

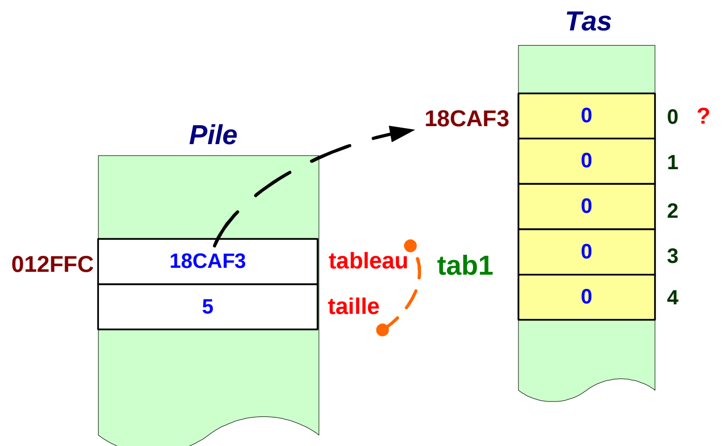
```cpp
10 TabInt::TabInt(int taille)
11 {
12   tableau = new int[this->taille = taille];
13   for (int i=0; i<taille; i++)
14     tableau[i] = 0;
15 }
```

Il est nécessaire de définir un constructeur qui récupère la taille demandée. Nous avons donc besoin d'un constructeur avec un paramètre. La dimension du tableau n'est connue qu'au moment où l'utilisateur en a besoin, c'est-à-dire au moment où il crée le nouvel objet. Dans ce contexte, la classe doit maîtriser une variable dynamique interne qui représente le tableau d'entiers. Nous en profitons pour initialiser chacune des cases à zéro.

Vu qu'il est impératif de créer cette variable dynamique, ce constructeur est obligatoire, ce qui fait que le constructeur par défaut est annihilé.

Comme il existe au moins une variable dynamique, il est également impératif de redéfinir le destructeur. En effet, il faut libérer la mémoire utilisée par la variable dynamique avant la destruction définitive de l'objet.

```cpp
17 TabInt::~TabInt()
18 {
19   delete[] tableau;
20 }
```



Redéfinition de l'opérateur « [ ] » pour la classe « TabInt »

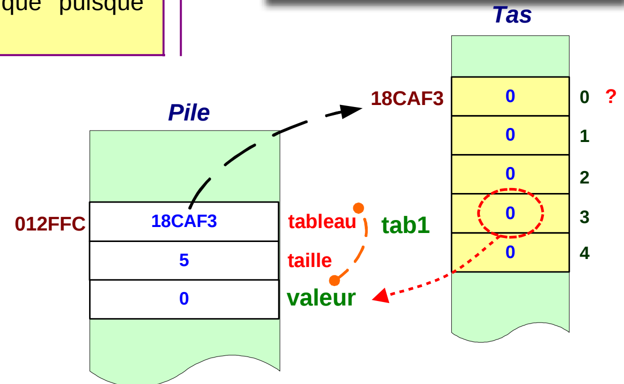
Pour permettre une utilisation sympathique de notre classe « *TabInt* », il est judicieux de proposer un fonctionnement au moins identique à un tableau classique, c'est-à-dire, de permettre l'accès à une des cases du tableau au moyen de crochets « *[ ]* » à l'intérieur desquels on spécifie le numéro de la case concernée.

La première approche que nous allons suivre consiste à récupérer une valeur d'une des cases du tableau. C'est le scénario qui vous est proposé ci-contre.

```
28 int main()
29 {
30   TabInt tab1(5);
31   int valeur = tab1[3];
32   return 0;
33 }
```

Remarquez bien que les crochets sont utilisés par rapport à l'objet tout entier. « tab1[3] » correspond à une valeur entière, ce qui est logique puisque « *tab1[3]* » représente une seule case du tableau d'entier.

```
1 class TabInt
2 {
3   int *tableau;
4   int taille;
5 public:
6   TabInt(int taille);
7   ~TabInt();
8   int operator[] (unsigned indice);
9 };
```



Puisque, au niveau de l'utilisation, les crochets sont utilisés en faisant référence à l'objet, il est alors nécessaire de définir l'opérateur « **[ ]** » en tant que méthode, c'est-à-dire, en tant que fonction membre. Pour respecter le contrat, l'opérateur doit renvoyer une valeur entière. Pour spécifier la case à atteindre, il est nécessaire d'utiliser un paramètre de type entier non signé.

L'implémentation interne est très réduite. Il suffit d'atteindre en interne la case concernée et de renvoyer la valeur correspondante.

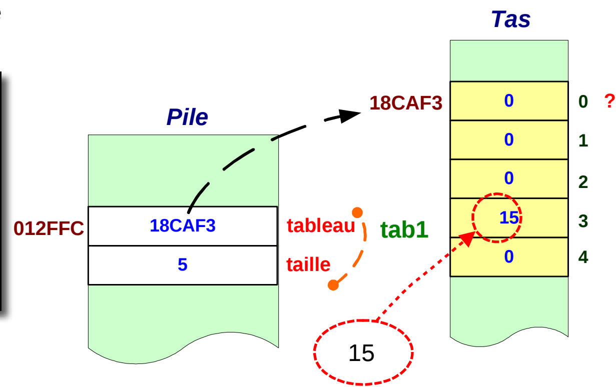
```
23 int TabInt::operator[] (unsigned indice)
24 {
25   return tableau[indice];
26 }
```

```
28 int main()
29 {
30   TabInt tab1(5);
31   tab1[3] = 15;
32   return 0;
33 }
```

Dans le scénario proposé ci-contre, il ne s'agit plus de récupérer une valeur, mais au contraire de proposer un changement directement sur la variable dynamique et dans la case concernée. La difficulté, c'est d'atteindre cette case. Il faut impérativement préciser l'adresse de la case à atteindre afin de pouvoir placer la valeur désirée. Du coup, le codage que nous avons proposé plus haut ne convient plus pour ce genre de situation.

```
1 class TabInt
2 {
3   int *tableau;
4   int taille;
5 public:
6   TabInt(int taille);
7   ~TabInt();
8   int& operator[] (unsigned indice);
9 };
```

Il est nécessaire de référencer la case à atteindre.




```
23 int& TabInt::operator[] (unsigned indice)
24 {
25   return tableau[indice];
26 }
```

La référence précise qu'on établit une connexion directement à « *tableau[indice]* ».

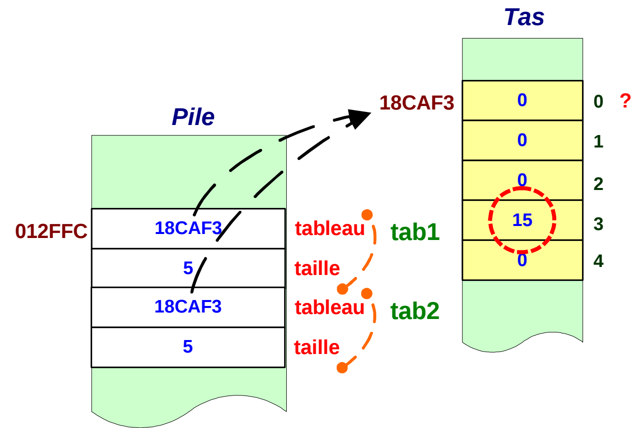
Rappel

Lorsque nous écrivons « *i = 12* », le point d'exécution sait qu'il doit atteindre une case mémoire dont l'adresse est fixée par la variable « i », et place dans cette case la valeur 12. Nous remarquons dans cette écriture, qu'une entité qui est placée à gauche du signe égal (***lvalue***), représente toujours une adresse. Soit, il existe une variable connue à cette adresse là, il est alors plus facile de désigner par son nom la variable concernée. C'est le cas de la variable « i ». Soit, la case mémoire n'a pas de nom spécifique (***anonyme***), à ce moment là, nous sommes obligés de prendre une référence. Remarquez, qu'il est également possible d'utiliser un pointeur, mais dans ce cas, ce serait un accès indirect.

Construction par copie

La classe que nous possédons actuellement parait séduisante. Toutefois, par acquis de conscience, nous allons tester tous les comportements par défaut. Nous allons, par exemple, proposer une construction par copie et visualiser ce qui se passe dans la mémoire.

```
28 int main()
29 {
30   TabInt tab1(5);
31   TabInt tab2 = tab1; // tab2(tab1)
32   tab1[3] = 15;
33   return 0;
34 }
```



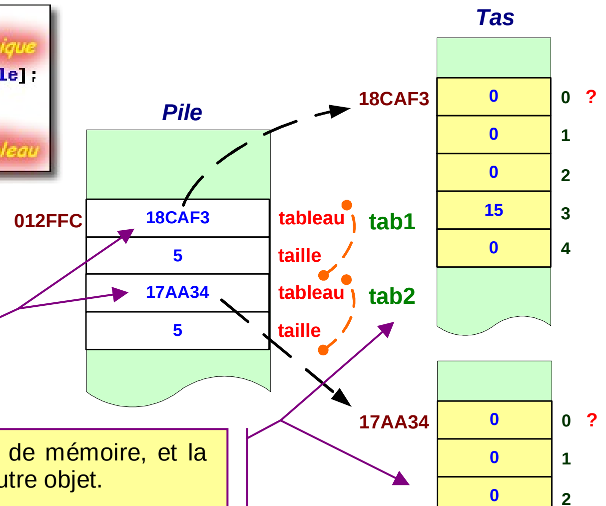
Le constructeur de copie est appelé au moment de la création de « ***tab2*** ». Par défaut, il exécute une copie de la valeur de chacun des attributs de « ***tab1*** ». Attention, c'est la seule copie effective. Si une variable dynamique existe, comme c'est le cas ici, elle ne fait pas partie de l'objet, elle n'est donc pas copiée. C'est un attribut qui y fait référence indirectement au moyen d'un pointeur. A ce sujet, n'oubliez pas que le contenu d'un pointeur est une adresse. Finalement, lorsque nous réalisons une copie d'un attribut de type pointeur, nous récupérons en fait son contenu - comme pour tous les attributs - mais le problème, c'est que nous récupérons une adresse et non pas le contenu de la variable pointée. Nous nous retrouvons donc avec deux objets distincts qui utilisent la même variable dynamique. En effet, l'attribut « ***tableau*** » de chacun des objets pointe vers la même adresse. Du coup, lorsque nous proposons une modification du contenu de la variable dynamique à l'aide de « ***tab1*** », indirectement cette modification se répercute sur « ***tab2*** ».

De façon encore plus dramatique, imaginons qu'un des objets, ***tab1*** ou ***tab2***, soit détruit avant l'autre. Lorsqu'il est effectivement détruit, il fait automatiquement appel au destructeur. Ce dernier libère la mémoire dynamique pointé par l'attribut ***tableau***. Malheureusement, le deuxième objet continu à pointer vers cette zone de mémoire qui vient d'être libérée. Si cette zone mémoire est utilisée par un autre type d'objet (ou d'ailleurs le même), imaginez un petit peu les conséquences.

Le comportement par défaut de la construction par copie n'est pas du tout adapté lorsque des objets gèrent des variables dynamiques. Il est alors nécessaire de redéfinir le comportement de la construction par copie pour que chaque objet dispose de sa propre variable dynamique et que la copie proposée soit plutôt une copie du contenu de la variable dynamique et non pas une copie de l'adresse.

```
1  class TabInt
2  {
3    int *tableau;
4    int taille;
5  public:
6    TabInt(int taille);
7    ~TabInt();
8    int& operator[] (unsigned indice);
9    TabInt(const TabInt&);
10 };
```

```
29 TabInt::TabInt(const TabInt& tab)
30 {                                  Nouvelle variable dynamique
31   tableau = new int[taille = tab.taille];
32   for (int i=0; i<taille; i++)
33     tableau[i] = tab.tableau[i];
34 }                                  Copie du tableau
```



Chaque objet dispose maintenant de sa propre adresse, c'est-à-dire, de sa propre variable dynamique.

Effectivement, chaque variable dynamique dispose de sa propre zone de mémoire, et la modification apportée sur une des zones n'a plus de répercutions sur l'autre objet.

Opérateur d'affectation

Nous imaginons bien également que l'affectation proposée par défaut n'est pas du tout adaptée à la situation puisqu'elle effectue également une copie membre à membre. Nous devons donc redéfinir ce comportement pour palier au problème. Puisqu'il est nécessaire de le redéfinir, nous en profitons pour voir deux cas d'utilisation. Après tout, c'est nous qui construisons cette classe et nous pouvons donc décider du comportement à atteindre. En effet, nous pouvons avoir :

- tab2 = tab1 ; *// une affectation simple et utiliser uniquement ce principe d'affectation simple.*
- tab3 = tab2 = tab1 ; *// une affectation multiple, ce qui est normalement l'utilisation par défaut prévu par le langage.*

L'opérateur d'affectation est un opérateur comme les autres. Du coup, l'opération simple peut également s'écrire de la façon suivante :

- tab2 = tab1; → tab2.operator=(tab1) ;

Si nous décidons d'utiliser uniquement l'affectation simple, nous remarquons qu'il n'est alors pas nécessaire de proposer un retour pour la méthode. Nous allons, dans un premier temps, proposer de redéfinir l'opérateur d'affectation pour ce cas de figure, ce qui nous simplifiera la tâche. Nous extrapolerons notre recherche pour implémenter ensuite l'affectation multiple. Ceci dit, nous pourrions nous contenter de l'affectation simple, puisque c'est essentiellement celle-ci qui est utilisée.

Nous allons avoir pratiquement le même codage que pour le constructeur de copie puisque par définition ils proposent la même attitude par défaut. Attention toutefois, pour l'affectation, il ne s'agit pas d'une création. En effet un tableau existe déjà, et il peut même avoir une taille différente. Ce n'est pas gênant, de toute façon, il faut détruire le tableau existant et récupérer une copie du tableau de l'objet passé en argument.

```
1  class TabInt
2  {
3    int *tableau;
4    int taille;
5  public:
6    TabInt(int taille);
7    ~TabInt();
8    int& operator[] (unsigned indice);
9    TabInt(const TabInt&);
10   void operator= (const TabInt&);
11 };
```

Sans retour

```
37 void TabInt::operator= (const TabInt& tab)
38 {
39   delete[] tableau;
40   tableau = new int[taille = tab.taille];
41   for (int i=0; i<taille; i++)
42     tableau[i] = tab.tableau[i];
43 }
```

Destruction de l'ancien tableau

Identique à la construction de copie

Dans le cas d'une affectation multiple, il faut proposer en plus de retourner un « ***TabInt*** », et pour que le système soit performant en terme de temps de réponse, il est préférable de proposer une référence pour éviter des copies supplémentaires.

- tab3 = tab2 = tab1; → tab3.operator=(tab2.operator=(tab1)) ;

```
1  class TabInt
2  {
3    int *tableau;
4    int taille;
5  public:
6    TabInt(int taille);
7    ~TabInt();
8    int& operator[] (unsigned indice);
9    TabInt(const TabInt&);
10   TabInt& operator= (const TabInt&);
11 };
```

```
37 TabInt& TabInt::operator= (const TabInt& tab)
38 {
39   delete[] tableau;
40   tableau = new int[taille = tab.taille];
41   for (int i=0; i<taille; i++)
42     tableau[i] = tab.tableau[i];
43   return *this;
44 }
```

Conclusion

Le fait d'utiliser des variables dynamiques dans une classe impose obligatoirement:

1. de mettre en place un constructeur pour permettre la création de la variable dynamique,
2. de redéfinir le destructeur pour être sûr que la zone mémoire utilisée par la variable dynamique soit libérée lorsque l'objet n'existe plus.
3. de redéfinir ou d'empêcher la construction de copie pour éviter la copie membre à membre puisqu'elle n'est plus adaptée à la situation.
4. de redéfinir ou d'empêcher l'affectation pour éviter la copie membre à membre puisqu'elle n'est également pas adaptée à la situation.

Blocages de la construction de copie et (ou) de l'affectation

Si vous ne désirez pas redéfinir la construction de copie ainsi que l'affectation, vous devez faire en sorte que ces méthodes ne puissent être utilisées, sinon il risque d'y avoir des aléas de fonctionnement. Pour réaliser le blocage, le principe est simple, il suffit de proposer la déclaration de ces méthodes en les mettant en zone privée. De cette manière, lorsque l'utilisateur tentera d'utiliser une de ces méthodes, le compilateur provoquera une erreur de compilation.

Avec le ***C++11***, il existe une manière plus élégante de réaliser le même type de blocage. Il suffit pour cela de spécifier le mot réservé « ***delete*** » à la suite de la méthode. En réalité, nous pouvons appliquer la spécification « ***= delete*** » à n'importe quelle méthode de la classe. Cela peut être utile, par exemple, afin d'empêcher l'utilisation d'une méthode particulière qui possède certains types de paramètres, et par contre autoriser une méthode sur-définie ne possédant pas ces paramètres à proscrire.

Blocage de la construction par copie et de l'affectation

```
class TabInt
{
  int *tableau;
  unsigned taille;
public:
  TabInt(unsigned taille);
  ~TabInt();
  int& operator[](unsigned indice);

  TabInt(const TabInt&) = delete;
  TabInt& operator=(const TabInt&) = delete;
};
```