La construction personnalisée des objets est très fréquente alors que la destruction l'est beaucoup moins. Nous verrons, malgré tout, des cas où il est absolument nécessaire de redéfinir le destructeur proposé par défaut. Avant cela, nous allons nous intéresser au pointeur « this ».

Pointeur « this »

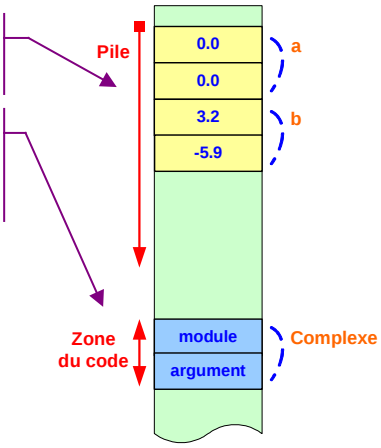
Ce qui est très sympathique dans la programmation des objets, c'est que les méthodes de la classe accèdent directement aux attributs de l'objet sans spécifications supplémentaires. Malgré tout, la question se pose de savoir comment fait la méthode pour arriver à se connecter sur les attributs de l'objet désiré. En effet, chaque objet de la classe possède sa propre copie des attributs, afin que chacun puisse avoir sa propre identité pour que leur état puisse évoluer de façon différente. Par contre, le comportement demeure identique pour tous les objets de la classe. En effet, il n'existe qu'une seule copie de chaque méthode. Chaque méthode doit donc fonctionner pour tous les objets de la classe. Examinons le scénario suivant :

```
//-----------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;
};
//-----------------------------------------
void fonction()
{
  Complexe a, b(5.6, 3.2);
  ...
  double nombre = b.module();
  ...
}
//-----------------------------------------
```

Chaque objet possède sa propre zone mémoire sur la pile.

Par contre, une seule définition des méthodes existe pour tous les objets.

Comment la méthode '***module***' sait qu'elle doit récupérer la valeur des attributs '***reel***' et '***imaginaire***' de l'objet « ***b*** » ?



Effectivement, comment fait la méthode '***module***' pour reconnaître qu'elle est associée à l'objet « ***b*** » et ainsi qu'elle peut travailler directement avec les attributs de « ***b*** » ? En fait, chaque méthode d'une classe possède un pointeur implicite (donc non visible à priori) qui adresse l'objet pour lequel la fonction membre est appelée. Ce pointeur est appelé « ***this*** ». Son type dans une méthode est un pointeur vers le type de la classe. Ainsi « ***this*** » pour la classe « ***Complexe*** » aura pour type : « ***Complexe**** ».

Dans les faits, le compilateur propose deux transformations pour incorporer ce pointeur afin de résoudre la connexion entre la méthode et les objets associés, à savoir :

1. D'abord, traduire la méthode pour q'elle possède un paramètre supplémentaire - le pointeur « ***this*** ».

```
//-----------------------------------------
inline double Complexe::module() const
{
  return sqrt(reel*reel + imaginaire*imaginaire);
}
//-----------------------------------------
```

```
//-----------------------------------------
inline double Complexe::module(Complexe *this) const
{
  return sqrt(this->reel * this->reel +
              this->imaginaire * this->imaginaire);
}
//-----------------------------------------
```

2. Ensuite, traduire chaque appel de la méthode pour ajouter un argument supplémentaire – l'adresse de l'objet pour lequel la méthode est invoquée.

```
//-----------------------------
void fonction()
{
  Complexe a, b(5.6, 3.2);
  ...
  double nombre = b.module();
  ...
```

```
//-----------------------------
void fonction()
{
  Complexe a, b(5.6, 3.2);
  ...
  double nombre = module(&b);
  ...
```

Tout ce que nous venons de voir se fait implicitement, et généralement, nous n'avons pas besoin de faire référence au pointeur « ***this*** ». Toutefois, il existe des situations où son utilisation directe peut s'avérer intéressante.

Possibilité d'avoir les paramètres des méthodes qui portent le même nom que les attributs :

Il est souvent très difficile de faire un choix judicieux quant au nom d'un paramètre. De plus, à l'heure actuelle, la plupart des environnements de développement intégré proposent des systèmes d'aides automatiques qui, dès que vous tapez le nom de la méthode, l'ensemble des paramètres attendus apparaissent avec leur type et leur nom. Dans ce cas là, il est judicieux de proposer des noms évocateurs pour que l'utilisateur sache ce que la méthode attend comme arguments.

```
//-----------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  {
    reel = x;  imaginaire = y;
  }
  double module() const;
```

En reprenant l'exemple du nombre complexe, j'avais d'abord choisi comme paramètres du constructeurs, respectivement, les noms '***x***' et '***y***'. Bien que nous comprenons à quoi cela correspond, cela reste imprécis.

En effet, à la construction, le tout c'est de savoir si l'objet à besoin d'une partie réelle et d'une partie imaginaire. Il devient plus judicieux d'employer directement ces noms là. Du coup, la question qui vient immédiatement à l'esprit, c'est :

Est-ce qu'il ne va pas y avoir un conflit entre le nom du paramètre et son attribut équivalent ? La réponse est « non », chacun est dans une portée différente :

- Pour le paramètre la portée, c'est la méthode,
- Pour l'attribut, la portée, c'est la classe.

La difficulté, toutefois, c'est de pouvoir atteindre l'attribut. C'est à ce moment là que nous avons besoin de l'opérateur « ***this*** » puisque que c'est justement son rôle.

Rappel

Lorsque nous faisons référence à un identificateur, c'est toujours la variable dont la portée est la plus locale qui est prise en compte. Lorsque nous évoquons '***imaginaire***', c'est donc une référence au paramètre de la méthode puisque sa portée est plus immédiate que la portée de la classe.

```
//-----------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double reel = 0.0, double imaginaire = 0.0);
  {
    this->reel = reel;
    this->imaginaire = imaginaire;
  }
  double module() const;
```

Attribut Paramètre

Possibilité d'appeler plusieurs méthodes séquentiellement :

Imaginons une classe « ***Notes*** » qui permet de gérer un ensemble de notes d'élèves, pour qu'à la suite, il soit possible de calculer : la moyenne, la note la plus haute, etc. Cette classe est composée d'un tableau de notes ainsi qu'un autre attribut qui gère à chaque instant, le nombre de notes déjà introduites. C'est la méthode « ***ajout*** » qui s'occupe d'insérer les notes successivement. Voici un scénario possible :

```
class Notes {
  double notes[10];
  unsigned nombreNotes;
public:
  Notes() { nombreNotes = 0; }
  void ajout(double note) {
    if (nombreNotes<NombreNotesMax)
      notes[nombreNotes++] = note;
  }
  void enlever();
  double moyenne() const;
  double plusPetite() const;
  double plusGrande() const;
  double lireNote(unsigned numero) const;
  unsigned short nombre() const;
};
```

Ce qui serait agréable, ce serait d'appeler plusieurs fois cette méthode « ***ajout*** », mais cette fois-ci, sur une seule ligne.

```
int main()
{
  Notes notes;
  notes.ajout(13.5);
  notes.ajout(8);
  notes.ajout(6.5);
  ...
}
```

```
int main()
{
  Notes notes;
  notes.ajout(13.5).ajout(8).ajout(6.5);
  ...
}
```

Attention, pour écrire de nouveau « ***ajout*** » après le séparateur « **.** », il est nécessaire d'avoir à gauche du séparateur, le type correspondant, c'est-à-dire, la classe « ***Notes*** », puisque cette méthode « ***ajout*** » n'existe que par rapport à cette classe. Il faut donc que cette méthode renvoie une référence vers sa propre classe (une référence permet de se connecter sans faire de copie). Finalement, dans la méthode « ***ajout*** », il faut employer un ***return*** et renvoyer l'objet dans lequel nous sommes. L'opérateur « ***this*** » est justement un pointeur vers l'objet sur lequel nous travaillons. Il est nécessaire de le déréférencer pour que l'on passe l'objet et non pas l'adresse de l'objet.

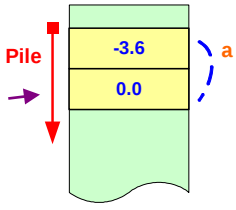
```
class Notes {
  double notes[10];
  unsigned nombreNotes;
public:
  Notes() { nombreNotes = 0; }
  Notes& ajout(double note) {
    if (nombreNotes<NombreNotesMax)
      notes[nombreNotes++] = note;
    return *this;
  }
  void enlever();
```

Destruction des objets

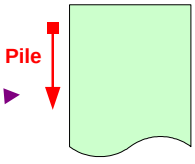
Comment se passe la destruction des objets ? Reprenons la classe « ***Complexe*** » afin de visualiser à la fois, la création et la destruction de l'objet.

```
void fonction()
{
  Complexe a = -3.6;
    ...
}
```

Création d'un objet qui est une variable automatique, donc elle se situe sur la pile. La durée de vie correspond à la fonction.



La fonction se termine, donc toutes les variables locales doivent être détruites. C'est le cas de notre objet. Comme, il s'agit d'une variable automatique comme une autre, elle est libérée automatiquement à la sortie de la fonction.

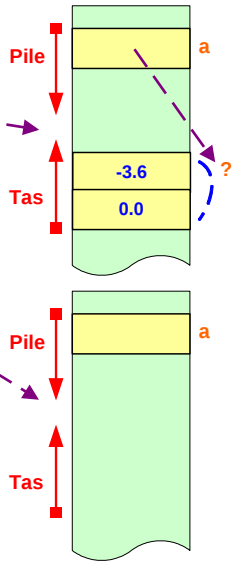


Finalement, tout se passe correctement. L'objet se détruit automatiquement quand il le faut. Justement, il est assez fréquent d'avoir plutôt des objets locaux. Toutefois, nous avons également besoin d'objets dynamiques. Cette fois-ci, cela ne se fait plus automatiquement, c'est le programmeur qui décide quand l'objet doit être créé, à l'aide de l'opérateur « ***new*** », et quand il doit être détruit, cette fois-ci grâce à l'opérateur « ***delete*** ».

```
void fonction()
{
  Complexe *a = new Complexe(-3.6);
    ...
  delete a;
    ...
}
```

Création d'un objet dynamique. Le pointeur reste une variable automatique. Elle est donc sur la pile. L'objet, lui, est créé sur le tas, puisque c'est un objet dynamique (également anonyme).

La destruction de l'objet s'effectue quand le programmeur le décide. La mémoire utilisée pour l'objet est alors libérée. Par contre le pointeur « ***a*** » existe toujours puisqu'il s'agit d'une variable automatique. Sa destruction ne se produira qu'à la sortie de la fonction.



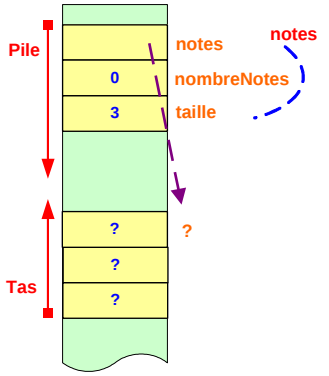
Nous avons découvert qu'il existait un destructeur. Vu ce que nous venons de voir, à quoi peut-il bien servir, puisque qu'apparemment tout se passe convenablement ?

Rôle du destructeur :

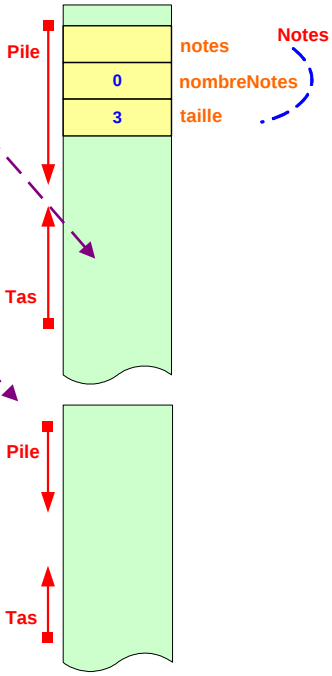
Le regard doit se porter à l'intérieur de la classe. Tout dépend de la déclaration des attributs, ou plutôt de la zone mémoire utilisée pour stocker leurs valeurs. Si les valeurs des attributs se situent sur la pile ou dans la zone statique, nous venons de voir que tout se gère automatiquement. Il existe un cas où le programmeur doit s'occuper de la destruction, c'est lorsqu'au moins un attribut fait référence à un objet dynamique. En effet, dans ce cas là, il sera nécessaire de libérer la variable dynamique avant que les attributs ne soient détruits automatiquement. C'est donc là que le destructeur intervient. Pour illustrer ces propos, reprenons l'exemple de la classe « ***Notes*** » en envisageant, cette fois-ci, que la taille du tableau est totalement variable. C'est au moment de la création de l'objet que l'on précise la dimension.

```
class Notes {
  double *notes;        Pointeur de double
  unsigned nombreNotes; Nouvel attribut
  unsigned taille;      pour gérer la dimension
public:
  Notes(unsigned taille) {
    nombreNotes = 0;
    notes = new double[this->taille = taille];
  }                     Création du tableau dynamique
Destructeur
  ~Notes()  { delete[] notes; }
                        Destruction du tableau dynamique
```

```
void fonction()
{
  Notes notes(3);
    ...
}
```



A la sortie de la fonction, l'objet « ***notes*** » doit être détruit. Au préalable, le destructeur est appelé, et la zone mémoire correspondant au tableau dynamique est libérée. Ce n'est qu'après que le destructeur est fait son travail que les attributs sont à leurs tours détruits.



Quand avons-nous besoin de redéfinir un destructeur ?

La réponse est systématiquement lorsque nous utilisons des variables dynamiques au sein de l'objet. Souvenez-vous que lorsque nous employons un « ***new*** », il faut qu'à un moment donné, il y ait un « ***delete*** ». Si c'est dans l'objet que nous avons écrit un « ***new*** », c'est également dans l'objet que nous devons écrire un « ***delete*** ».

Par ailleurs, dès que vous avez un pointeur comme attribut, posez-vous la question de savoir si nous n'avons pas besoin de redéfinir le destructeur.