Prise de connaissance entre les allocations statiques et dynamiques

En C++ les variables peuvent être allouées de deux façons différentes :

1. Soit **statiquement** : Avant l'exécution du programme, le compilateur traite le code source et détermine l'ensemble des variables et réserve un emplacement mémoire en conséquence. La durée de vie de ces variables correspond à la durée de vie du programme. C'est ce type d'allocation que nous avons réalisé jusqu'à présent. La zone d'allocation est soit la zone statique, soit la pile (pour les variables locales). Les zones d'allocations seront traitées ultérieurement.

2. Soit **dynamiquement** : Cette fois-ci, pendant l'exécution du programme, il est possible d'avoir besoin d'une variable pour une utilisation relativement brève dans le temps. Une variable est alors créée à la volée et elle sera détruite quand le besoin ne s'en fera plus sentir. La zone d'allocation est le tas.

Le compromis entre les deux méthodes d'allocation mémoire est l'efficacité contre la flexibilité :

- L'allocation de mémoire statique est considérablement **plus efficace** car effectué avant le commencement du programme. Quand le programme démarre, tout est prêt pour fonctionner correctement et les variables sont très faciles à atteindre et à manipuler. Par ailleurs le temps de réponse est des plus rapide.

- Elle est toutefois **moins flexible** car elle nécessite de connaître, avant l'exécution du programme, la quantité et le type de mémoire désirés. La taille mémoire nécessaire pour tout le temps d'exécution du programme peut être très conséquent suivant le nombre de variables. Par ailleurs, il n'est pas sûr qu'une variable soit utile pendant toute la durée de vie du programme. Imaginons, par exemple, que nous devions construire un logiciel avec une bonne interface IHM (Interface Homme Machine). Ce logiciel doit utiliser vingt fenêtres différentes, seulement, on est sûr que pendant le déroulement du programme, seules trois sont utilisées en même temps. Avec une allocation statique, il y aurait un gros gaspillage de mémoire. C'est vraiment le cas de figure ou l'allocation dynamique est prépondérante. Nous créons les fenêtres juste au moment où nous en avons besoin.

Détail des structures entre les allocations statiques et dynamiques

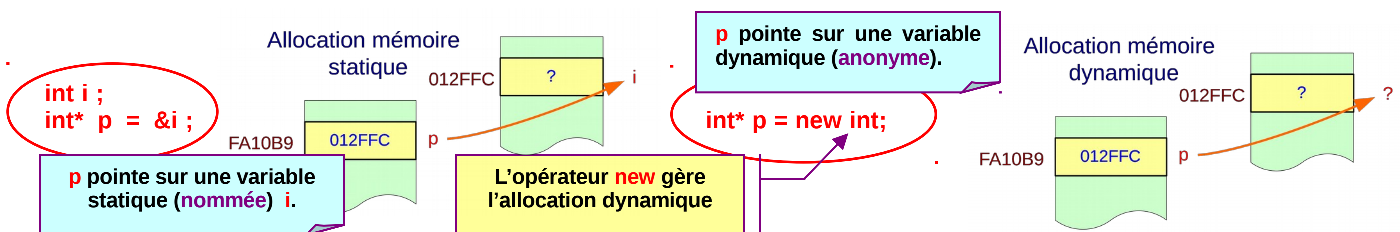
Toutes les déclarations réalisées jusqu'à présent étaient essentiellement pour des variables statiques. Les deux premières différences entre allocations de mémoire statique et dynamique sont les suivantes :

- Les variables statiques sont des variables nommées que l'on manipule directement, alors que les variables dynamiques sont des variables non nommées (**anonymes**) manipulées indirectement au travers de pointeurs.
- L'allocation et la désallocation de variables statiques sont gérées automatiquement par le compilateur ; l'allocation se fait au démarrage du programme et la désallocation à la fermeture de ce dernier. Le programmeur doit comprendre ce qui se passe, mais, il n'a rien à faire de spécial à ce sujet. L'allocation et la désallocation de variables dynamiques, en revanche, doivent être explicitement gérées par le programmeur et, en pratique, sont plus susceptibles de générer des erreurs. Ceci s'effectue grâce à l'utilisation des opérateurs **new** et **delete**.

Allocation dynamique

Sur un type quelconque d'une unité :

Il faut donc utiliser les pointeurs pour gérer l'allocation dynamique puisque l'accès se fait de façon indirecte. Jusqu'à présent, les pointeurs étaient essentiellement utilisés pour se connecter indirectement sur une autre variable statique et éventuellement pour naviguer au travers de celle-ci – grâce à l'arithmétique des pointeurs - si cette dernière représente un tableau.



L'opérateur **new** s'occupe de l'allocation dynamique. Pour qu'il soit à même de réserver un emplacement suffisant, il faut lui indiquer le type correspondant. Le type indiqué avec l'opérateur **new** doit être de même nature que le type géré par le pointeur. La syntaxe générale sur l'opérateur **new** est donc la suivante :

```
Type* pointeur ;
pointeur = new Type ;
      ou
Type* pointeur = new Type ;
```

Les types sont adaptés

Type représente n'importe quel type primaire comme **int**, **double**, etc., mais également des types définis par les utilisateurs comme les **structures**, les **unions**, les **énumérations**, mais pas les tableaux. Pour ces derniers, nous aurons une syntaxe adaptée.

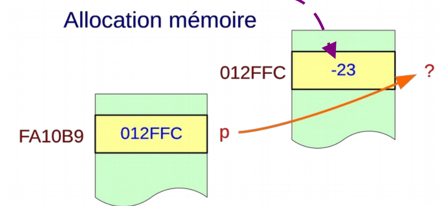
```
//-----------------------------------------------------------------
enum CouleurCarte {Pique, Coeur, Carreau, Trefle};
enum ValeurCarte {As, Roi, Dame, Valet, Dix, Neuf, Huit, Sept};
//-----------------------------------------------------------------
struct Carte {
  CouleurCarte couleur;
  ValeurCarte valeur;
};
//-----------------------------------------------------------------
union Reel {
  float nombre;
  struct {
    unsigned mantisse : 23;
    unsigned exposant : 8;
    int signe : 1;
  }
};
//-----------------------------------------------------------------
int main()
{
  int *pInt = new int;              // allocation dynamique d'un entier
  double *pDouble = new double;     // allocation dynamique d'un réel
  CouleurCarte *pCouleur = new CouleurCarte;    // " d'une énumération
  Carte *pCarte = new Carte;    // allocation dynamique d'une structure
  Reel *pReel = new Reel;       // allocation dynamique d'une union
  ...
  return 0;
}
```

Quelques exemples

Initialisation dans la mémoire allouée :

Il est possible pour les types primitifs de donner une valeur initiale à la valeur pointée. Il suffit d'utiliser les parenthèses associé au type en donnant comme paramètre la valeur initiale désirée, comme suit :

int* p = new int (-23) ;



A partir de nos connaissances actuelles, il n'est pas possible d'initialiser explicitement des types définis par l'utilisateur comme les structures, les énumérations et les unions. Il sera nécessaire de connaître la notion de constructeur.

```
//-----------------------------------------
struct Personne {
  char *nom;
  int age;
};
//-----------------------------------------
int main()
{
  Personne *personne = new Personne;
  personne->nom = "Durand";
  personne->age = 25;
  // ...
  delete personne;
  return 0;
}
```

Utilisation des variables dynamiques :

En fait, il n'y a pas de remarque particulière sauf qu'il faut se souvenir qu'une variable dynamique est **anonyme**, et donc que la seule possibilité de l'atteindre est d'utiliser systématiquement le pointeur associé.

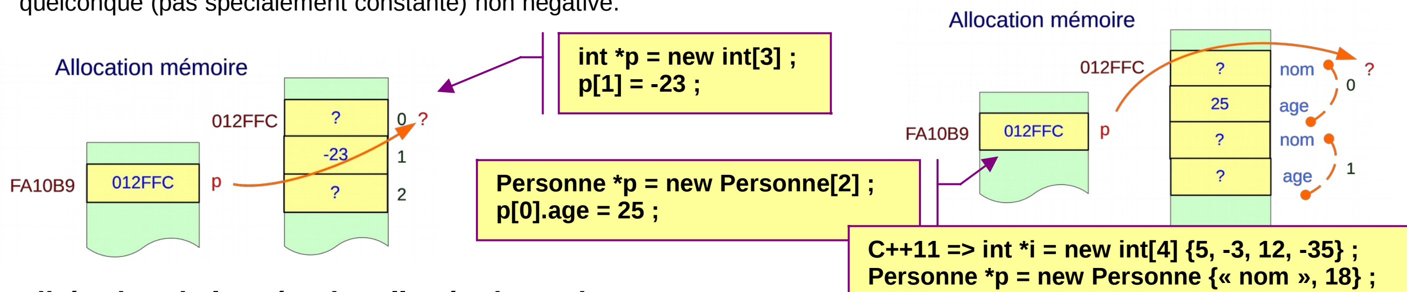
Souvenez-vous que pour les variables statiques, lorsque nous déclarions un pointeur, il référençait une autre variable statique. Il était alors possible, soit d'utiliser directement la variable pointée, soit indirectement par le pointeur. Dans le cas d'une variable dynamique, la question ne se pose pas, nous sommes systématiquement obligés d'utiliser le pointeur.

Les tableaux dynamiques :

L'opérateur **new** permet également de déclarer des variables tableaux dynamiques d'un type quelconque. Malheureusement, il n'existe aucun moyen de spécifier individuellement une valeur initiale explicite pour chaque élément du tableau (sauf pour le ***C++11***). La syntaxe générale est la suivante :

Type* pointeur = new Type[n] ;

Cette instruction alloue alors l'emplacement nécessaire pour n éléments du type spécifié ; si l'opération réussit (suffisamment de mémoire), elle fournit alors au pointeur l'adresse du premier élément de ce tableau. « n » désigne une expression entière quelconque (pas spécialement constante) non négative.



C++11 => int *i = new int[4] {5, -3, 12, -35} ;
Personne *p = new Personne {« nom », 18} ;

Libération de la mémoire allouée dynamiquement

Il est impératif de libérer la mémoire lorsque nous avons fini de nous servir d'une variable dynamique. Sinon, tout le principe que nous venons d'évoquer ne servirait absolument à rien. La mémoire libérée peut servir ensuite pour les autres variables dynamiques. Ainsi, il est possible, avec peu de mémoire, d'utiliser énormément de variables différentes. Par ailleurs, n'oubliez pas que pour un système multitâche, les autres applications ont besoin elles-mêmes de mémoires pour fonctionner, la mémoire étant partagée pour tous les processus.

Pour libérer une mémoire allouée dynamiquement, il suffit d'utiliser l'opérateur **delete**. Toutefois, cet opérateur a besoin de connaître l'emplacement de la mémoire à libérer. Comme elle est anonyme, encore une fois, il est nécessaire de passer par le pointeur. Pour libérer un tableau, il suffit de placer les crochets à côté de l'opérateur **delete** sans précision de la dimension.

Int* pint = new int ;
Personne* pPersonne = new Personne ;
…
delete pint ;
delete pPersonne ;

A chaque **new** doit correspondre un **delete** associé

int* pTabInt = new int[27] ;
Personne* pTabPersonne = new Personne[124] ;
…
delete [ ] pTabInt ;
delete [ ] pTabPersonne ;

Méthodes utiles de la classe vector<>

La classe ***vector<>*** remplace aisément les tableaux classiques en offrant des manipulations simples et intuitives. Il est possible d'initialiser un tableau avec une valeur particulière pour toutes les cases du tableau ou même de spécifier une valeur d'initialisation différente pour chacune des cases du tableau grâce aux listes d'initialisation.

Avec cette classe vector<>, certaines opérations peuvent être réalisées simplement :

1. ***=*** *: l'affectation est possible entre deux tableaux de même type.*
2. ***[ ]*** *: l'opérateur d'indexation à bien évidemment été redéfini pour supporter le comportement classique d'un tableau, puisque cet opérateur à été spécialement créé pour les tableaux.*
3. ***==, !=, <, <=, >, >=*** *: Il est de plus possible de comparer le contenu de deux tableaux entre eux en utilisant les opérateurs classiques de comparaison.*

Méthodes utiles

1. ***clear()*** *: vide le tableau de son contenu.*
2. ***empty()*** *: teste si le tableau est vide et renvoie **true** si c'est le cas, et **false** dans le cas contraire.*
3. ***swap()*** *: permet d'échanger le contenu de deux tableaux de même type.*
4. ***push_back(valeur)*** *: cette méthode est spécialisée pour insérer une valeur en fin de tableau à la manière d'une pile.*
5. ***pop_back()*** *: cette méthode est spécialisée pour supprimer la dernière valeur du tableau à la manière d'une pile.*
6. ***size()*** *: détermine le nombre d'éléments que contient le tableau.*
7. ***back()*** *: récupère la valeur du dernier élément sans toutefois l'enlever du tableau comme c'est le cas avec la méthode **pop_back()**.*
8. ***front()*** *: récupère la valeur du premier élément sans l'enlever du conteneur.*

Méthodes utiles de la classe string

Une variable de type ***string*** contient une suite formée d'un nombre quelconque de caractères. Sa taille peut évoluer dynamiquement au fil de l'exécution du programme. Si la chaîne désirée est plus grand que cette zone réservée, la classe augmente automatiquement ce bloc en proposant une nouvelle allocation mémoire et en prenant la précaution d'avoir un bloc plus grand que nécessaire afin de répondre rapidement à une petite augmentation de la taille de la chaîne. Enfin, la notion de caractère de fin de chaîne n'existe plus pour cette classe.

Opérateurs utilisables avec la classe string

1. ***=*** *: affecte une chaîne à une autre.*
2. ***[ ]*** *: joue le même rôle que pour une chaîne de caractères classique du C++.*
3. ***+=*** *: ajoute une chaîne à la fin d'une autre.*
4. ***+*** *: cet opérateur assure la concaténation de deux chaînes de caractères pour former une troisième chaîne.*
5. ***==, !=, <, >, <=, >=*** *: ces opérateurs renvoient **true** ou **false** suivant la comparaison qui est faite sur deux chaînes de caractères. Les différentes comparaisons évaluent en fait l'ordre alphabétique.*

Méthodes utiles

1. ***clear()*** *: vide entièrement la chaîne de caractères.*
2. ***c_str()*** *: cette méthode permet de passer d'une chaîne de type **string** vers une chaîne classique du C++ (**const char ***). Attention, il est possible de récupérer cette chaîne sans toutefois pouvoir la modifier puisque une constante est déclarée.*
3. ***empty()*** *: retourne **true** si la chaîne est vide (sans aucun caractère), sinon retourne **false** .*
4. ***erase()*** *: efface une partie de la chaîne ou un caractère spécifié en argument.*
5. ***find()*** *: effectuent des recherches sur une partie de la chaîne ou sur un caractère spécifié en argument.*
6. ***insert()*** *: permet d'insérer une autre chaîne ou bien un ou plusieurs caractères donnés.*
7. ***length()*** *: retourne la longueur de la chaîne de caractères. Similaire à la méthode **size()**.*
8. ***replace()*** *: remplace une partie de chaîne.*
9. ***size()*** *: retourne la longueur d'une chaîne de caractères. Similaire à **length()**.*
10. ***substr()*** *: retourne une partie de chaîne.*
11. ***swap()*** *: assure la permutation de deux chaînes de caractères.*

Fonctions utiles en relation avec la classe string pour la manipulation de nombres – C++11

Il est très fréquent d’avoir besoin de représenter des valeurs numériques sous forme de chaînes de caractères et inversement. Depuis le **C++11**, nous disposons d’une fonction simple qui permet de passer d’un nombre vers une chaîne et plusieurs fonctions pour l’inverse dont voici la liste :

Fonctions utiles associées à la classe string

1. ***to_string()*** *: renvoie la chaîne de caractères équivalente à la valeur numérique proposée en argument de la fonction.*
2. ***stoi()*** *: renvoie une valeur numérique entière de type **int** à partir de la chaîne passée en argument de la fonction.*
3. ***stol()*** *: renvoie une valeur numérique entière de type **long** à partir de la chaîne passée en argument de la fonction.*
4. ***stoul()*** *: renvoie une valeur numérique entière de type **unsigned int** à partir de la chaîne passée en argument de la fonction.*
5. ***stof()*** *: renvoie une valeur numérique réelle de type **float** à partir de la chaîne passée en argument de la fonction.*
6. ***stod()*** *: renvoie une valeur numérique réelle de type **double** à partir de la chaîne passée en argument de la fonction.*

Expériences en utilisant le débogueur sur les variables dynamiques, les classes vector et string

main.cpp

```
#include <iostream>  // Tester en c++11
#include <string>
#include <vector>
using namespace std;

struct Complexe
{
  double reel, imaginaire;
};

struct Eleve
{
  string nom, prenom;
  vector<double> notes;
};

vector<string> prenoms;

int main()
{

  Complexe a = {5.3, -6.4};
  Complexe* p = &a;
  Complexe* b = new Complexe {3.2, 9.8};

  a.reel = 12.0;
  p->imaginaire = -3.0;
  b->reel = -2.5;
  b->imaginaire = a.imaginaire;
  *b = {3.2, 7.8};
  a = *b;

  Eleve toi = {"MERCIER", "Fabien"};
  toi.notes.push_back(10.0);
  toi.notes.push_back(8.5);
  toi.notes.push_back(12.0);
  toi.notes.push_back(15.5);

  Eleve moi = toi;
  moi.nom = "DURANT";
  moi.prenom = "Michel";

  prenoms = {"alain", "michel", "bruno", "alice"};
  prenoms.push_back(toi.prenom);
  prenoms.push_back(moi.prenom);
  toi.prenom = prenoms[0];

  for (int i=0; i<prenoms.size(); i++) if (prenoms[i][0]>='a' && prenoms[i][0]<='z') prenoms[i][0] -= 0x20;
  // ou
  for (string &prenom : prenoms) if (prenom[0]>='a' && prenom[0]<='z') prenom[0] -= 0x20;

  cout << "prénoms = [";
  for (string prenom : prenoms) cout << prenom << ' ';
  cout << "\b]\n";

  cout << toi.prenom << ' ' << toi.nom << " : [";
  for (double note : toi.notes) cout << note << ' ';
  cout << "\b]\nMoyenne = ";

  if (!toi.notes.empty())
  {
    double somme = toi.notes[0];
    for (int i=1; i<toi.notes.size(); i++) somme += toi.notes[i];
    cout << somme/toi.notes.size() << endl;
  }

  prenoms.clear();
  if (prenoms.empty()) cout << "Plus de prénoms" << endl;

  return 0;
}
```