



Initiation au langage

Objet C++

COMPLEMENTS

TABLE DES MATIERES

1) Rappels - Flux standards les plus communs

1.1) Les entrées - sorties page N°3
1.2) Les fonctions d'attente page N°5

2) Les allocations dynamiques

2.1) Introduction page N°6
2.2) Allocation dynamique d'une variable page N°6
2.3) Allocation dynamique d'un tableau page N°8

3) Les fonctions virtuelles et classes abstraites

3.1) Les fonctions virtuelles page N°10
3.2) Les classes abstraites page N°11

4) Annexes

4.1) Les classes abstraites page N°15
4.2) Le pointeur This page N°16
4.3) Diagramme de classes page N°18

Initiation au langage objet C++ - COMPLEMENTS

1) Rappels - Flux standards les plus communs

1.1) Les entrées – sorties

- Première chose, il faut inclure la bibliothèque d'entrées-sorties du C++, qui est « **iostream** ».

- Pour les **entrées**, on connaît le mot-clé « **cin** » en C++, équivalent au « **scanf( )** » du langage C. Ces fonctions sont pour la saisie clavier, à la seule différence que « scanf » attend qu'on lui spécifie le type (%..) de la valeur entrée au clavier, et qu'elle corresponde à la variable déclarée dans laquelle on enregistre cette valeur.

Au contraire, « cin » s'adapte au type de la variable déclarée dans laquelle on enregistre la valeur entrée au clavier.

Exemple « cin » :

```
#include < iostream >

int var;
cout <<"Entrer un nombre : ";
cin >>var;                          //Attend la saisie au clavier d'un nombre pour le mettre dans "var"
```

Exemple « scanf » :

```
#include < stdio.h >

int var;
printf("Entrer un nombre : ");
scanf("%d", &var);      //Attend la saisie au clavier d'un nbre de type INT  pour le mettre dans "var"
```

Affichage à l'écran pour les deux cas :



- Pour les **sorties**, on connaît le mot-clé « **cout** » en C++, équivalent au « **printf( )** » du langage C. Ces fonctions sont pour l'affichage dans la fenêtre de commande (d'exécution), à la seule différence que « printf » attend qu'on lui spécifie le type (%..) de la valeur affichée à l'écran.

Au contraire, « cout » s'adapte au type de la variable déclarée dont on veut afficher la valeur à l'écran.

Exemple « cout » :

```
#include < iostream >

int var;
var=3;
cout <<"La valeur de var est : "<< var;          //On affiche 3
```

Exemple « printf » :

```
#include < stdio.h >

int var;
var=3;
printf("La valeur de var est : %d", var);          //On affiche 3
```

Affichage à l'écran pour les deux cas :



Dans les **sorties**, on peut aussi utiliser le mot-clé « **cerr** », qui a exactement la même action que « cout ». On se sert de « cerr » pour faire la différence, dans la lecture d'un programme, entre l'affichage à l'écran d'un message d'erreur (on utilise « cerr »), et l'affichage de tout autre message.

Exemple « cout - cerr » :

```
#include < iostream >

cout <<"Traitement en cours.."<< endl;
cerr <<"ERREUR"<< endl;
```

Affichage à l'écran :



Pour finir, la terminaison facultative « **endl** » que l'on place en fin de « cout » ou « cerr » permet que le curseur d'écran revienne à la ligne après avoir affiché ce qu'on lui demande à l'écran.

1.2) Les fonctions d'attente

Selon les compilateurs et logiciels de codage, on a parfois besoin de mettre dans notre programme une fonction permettant de faire attendre la fenêtre d'affichage ("l'écran") pour qu'on puisse voir le résultat de notre programme. Car parfois les fenêtres s'ouvrent puis se ferment instantanément.

Pour palier à ce problème, on peut utiliser une fonction d'attente. Les deux fonctions : « **system("PAUSE");** » ou « **getch( );** » bloquent la fenêtre ouverte et attend que l'utilisateur appuie sur une touche.

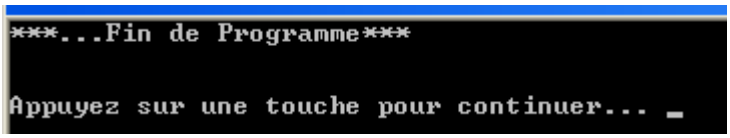
La différence entre les deux c'est que « system("PAUSE"); » écrit sur l'écran : « Appuyer sur une touche… » pour spécifier à l'utilisateur qu'il attend l'appuie d'une touche pour fermer la fenêtre. La fonction « getch( ); » n'écrit rien à l'écran, mais fonctionne similairement : dès qu'une touche est appuyée, il laisse la fenêtre d'affichage se fermer.

Exemple « system("PAUSE"); » :

```
#include < iostream >

cout <<"***...Fin de Programme***\n\n";
system("PAUSE");
```

Affichage à l'écran :

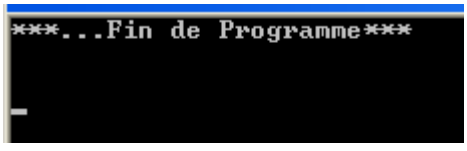


Exemple « getch( ); » :

```
#include < conio.h >
#include < iostream >

cout <<"***...Fin de Programme***\n\n";
getch( );
```

Affichage à l'écran :



Rem : Pour la fonction « **getch( )** », il faut inclure la bibliothèque « **conio.h** ».

2) Allocations dynamiques

2.1) Introduction

Rappels : On connaît en langage C les fonctions « malloc » et « free ».

Exemple – Code en C :

```
int main( )
{
        int *variable = NULL;
        variable = malloc(sizeof(int));          //Allocation de mémoire
        free(variable);                          //Libération de mémoire
        return 0;
}
```

L'allocation dynamique est une technique qui nous permet de gérer l'allocation de mémoire pour des variables. C'est notamment très pratique dans le cas de l'allocation de tableaux dont on ne connaît pas la taille avant compilation.

En C++, on fait toujours des allocations dynamiques. D'ailleurs, les fonctions « malloc » et « free » sont toujours utilisables. Cependant, le C++ dispose de nouveaux opérateurs spécialisés dans les allocations dynamiques : « new » et « delete ».

Les opérateurs **new** et **delete** sont des mots-clé dans le langage C++. Contrairement à **malloc** et **free**, ce ne sont pas des fonctions. Les opérateurs **new** et **delete** font en fait eux-mêmes appels aux fonctions **malloc** et **free** (on se sert de l'existant). Cependant, ils font des tests et des initialisations supplémentaires, ce qui fait qu'on préfèrera toujours utiliser **new** et **delete** au lieu de **malloc** et **free**. Ils sont plus adaptés en C++.

2.2) Allocation dynamique d'une variable

- Allocation de mémoire :

Les opérateurs **new** et **delete** ne s'utilisent pas exactement de la même manière que **malloc** et **free**.
On va dans un premier temps apprendre à s'en servir pour allouer une variable

simple, puis on verra ensuite le cas de l'allocation de tableaux.

On souhaite donc allouer dynamiquement une variable (de type **int** par exemple). En C++, on va d'abord devoir créer le pointeur et l'initialiser à NULL :

Exemple – Code en C++ - *new* :

```
int *variable = NULL;
variable = new int;              //Allocation dynamique
```

On n'a plus besoin d'utiliser l'opérateur de taille **sizeof( )** du C. Ici, on indique juste le type de variable à créer.

- Libération de mémoire :

Lorsqu'on a fini d'utiliser notre variable et qu'on en a plus besoin, vous devez la libérer avec l'opérateur **delete**.

Exemple – Code en C++ - *delete* :

```
delete variable;                 //Libération de mémoire
```

- Résumé :

new et **delete** étant des opérateurs, et non des fonctions, on ne met pas de parenthèses.

Le programme ci-dessous résume un exemple complet de l'allocation / libération de mémoire en C++ :

Exemple – Code en C++ - *new et delete* :

```
int main( )
{
        int *variable = NULL;
        variable = new int;                 //Allocation de mémoire
        delete variable;                    //Libération de mémoire
        return 0;
}
```

2.3) Allocation dynamique d'un tableau

- Allocation de mémoire :

Si on veut allouer un tableau, on n'a plus besoin de faire un calcul du type **20*sizeof(int)** comme on devait le faire en langage C. On va utiliser l'opérateur **new[ ]**.
On commence par créer le pointeur, puis on alloue la mémoire :

Exemple – Code en C++ - *new[ ]* :

```
int *tableau = NULL;
tableau = new int[20];          //Allocation de mémoire (20 cases)
```

Dans ce cas, un tableau de 20 cases sera alloué. Bien entendu, il est aussi possible de remplacer ce nombre par une variable :

Exemple – Code en C++ - *new[ ]* :

```
int *tableau = NULL;
tableau = new int[taille];      //Allocation de mémoire ("taille" cases)
```

La longueur du tableau sera définie par la valeur de la variable taille.

- Libération de mémoire :

Lorsqu'on n'a plus besoin du tableau, on doit le libérer... avec cette fois l'opérateur **delete[ ]** pour bien préciser qu'il s'agit d'un tableau. On n'a pas besoin de préciser la taille entre crochets, mais il ne faut pas oublier de les mettre quand même, c'est important pour le compilateur.

Exemple – Code en C++ - *delete[ ]* :

```
delete[ ] tableau;              //Libération de mémoire
```

- Résumé :

Le programme ci-dessous résume un exemple complet de l'allocation / libération de mémoire pour les tableaux en C++ :

Exemple – Code en C++ - *new[] et delete[]* :

```
int main( )
{
        int *tableau = NULL;
        tableau = new int[20];          //Allocation de mémoire (tableau)
        delete[ ] tableau;              //Libération de mémoire (tableau)
        return 0;
}
```

Il y a donc en tout 4 opérateurs :

- **new** s'utilise avec **delete** pour allouer une variable.
- **new[ ]** s'utilise avec **delete[ ]** pour allouer un tableau.

3) Les fonctions virtuelles et classes abstraites

3.1) Les fonctions virtuelles

Soit **f** une fonction membre d'une classe nommée **C**. Si les conditions suivantes sont réunies :

- **f** est redéfinie dans des classes dérivées (directement ou indirectement) de **C**,
- **f** est souvent appelée à travers des pointeurs ou des références sur des objets de la classe **C** ou de classes dérivées de **C**,

=> Alors **f** mérite d'être une **fonction virtuelle**. On exprime cela en faisant précéder sa déclaration du mot réservé « **virtual** ».

L'important service obtenu est celui-ci : si **f** est appelée à travers un pointeur ou une référence sur un objet de la classe **C**, le choix de la fonction effectivement activée, parmi les diverses redéfinitions de **f**, se fera d'après le type dynamique de cet objet.

Une classe possédant des fonctions virtuelles est dite classe « polymorphe ». La qualification **virtual** devant la redéfinition d'une fonction virtuelle est facultative : les redéfinitions d'une fonction virtuelle sont virtuelles d'office.

Exemple – Code en C++ : (**f** équivaut à la fonction ***seDessiner*** et **C** équivaut à la classe ***Figure***)

```
class Figure
{
        public:
                virtual void seDessiner( )
                {
                        // fonction passe-partout, probablement sans grand intérêt
                }
                ...
};

class Triangle : public Figure
{
        public:
                void seDessiner( )
                {
                        // implémentation du tracé d'un triangle
                }
                ...
};

class Ellipse : public Figure
```

```
{
        public:
                void seDessiner( )
                {
                        // implémentation du tracé d'une ellipse
                }
                ...
};

Figure *image[maxFigure];
int n = 0;
image[n++] = new Triangle(...);
image[n++] = new Figure(...);
image[n++] = new Ellipse(...);
...
for (int i = 0; i < N ; i++) image[i]->seDessiner( );
```

Maintenant, l'appel **image[i]->seDessiner( )** active la fonction redéfinie dans la classe de l'objet effectivement pointé par **image[i]**. Cela constitue, bien entendu, le comportement intéressant, celui qu'on recherche presque toujours. On notera que cet appel est légal parce que la fonction **seDessiner** a été déclarée une première fois dans la classe qui correspond au type statique de **image[i]**.

3.2) Les classes abstraites

Les fonctions virtuelles sont souvent introduites dans des classes placées à des niveaux si élèvés de la hiérarchie (d'héritage) qu'on ne peut pas leur donner une implémentation utile ni même vraisemblable.

Une première solution consiste à en faire des fonctions vides :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( )
                {
                }       // sera définie dans les classes dérivées
                ...
};
```

En fait, si la fonction **seDessiner** peut être vide c'est parce que :

- La classe **Figure** ne doit pas avoir d'objets ;
- La classe **Figure** doit avoir des classes dérivées (qui, elles, auront des objets) ;
- La fonction **seDessiner** doit être redéfinie dans les classes dérivées.

Bref, la classe **Figure** est une *abstraction*. Un programme ne créera pas d'objets **Figure**, mais des objets **Rectangle**, **Ellipse**, etc., de classes dérivées de **Figure**. Le rôle de la classe Figure n'est pas de produire des objets, mais de représenter ce que les objets Rectangles, Ellipses, etc., ont en commun. Or, s'il y a une chose que ces objets ont en commun, c'est bien la faculté de se dessiner. Nous dirons que ***Figure est une classe abstraite.***

On remarque que la fonction **seDessiner** introduite au niveau de la classe **Figure** n'est pas un service rendu aux programmeurs des futures classes dérivées de Figure, mais une contrainte : son rôle n'est pas de dire ce qu'est le dessin d'une figure, mais d'obliger les futures classes dérivées de Figure à le dire.

On peut traduire cet aspect de la chose en faisant que cette fonction déclenche une erreur. En effet, si elle est appelée, c'est que ou bien on a créé un objet de la classe Figure, ou bien on a créé un objet d'une classe dérivée de Figure dans laquelle on n'a pas redéfini la fonction seDessiner :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( )
                {
                        erreur("on a oublie de redéfinir la fonction seDessiner");
                }
                ...
};
```

Cette manière de faire est peu pratique, car l'erreur ne sera déclenchée qu'a l'exécution (c'est le programmeur qui a commis l'oubli mais c'est l'utilisateur qui se fera réprimander !).

En C++ on a une meilleure solution :

Définir une **fonction virtuelle pure**, par la syntaxe :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( ) = 0;          // Une fonction virtuelle pure
                ...
};
```

D'une part, cela officialise le fait que la fonction ne peut pas, à ce niveau, posséder une implémentation.

D'autre part, cela crée, pour le programmeur, l'obligation de définir cette implémentation dans une classe dérivée ultérieure, et permet de vérifier pendant la compilation que cela a été fait. Une fonction virtuelle pure reste virtuelle pure dans les classes dérivées, aussi longtemps qu'elle ne fait pas l'objet d'une redéfinition (autre que « = 0 »).

Pour le compilateur, une classe abstraite est une classe qui a des fonctions virtuelles pures. Tenter de créer des objets d'une classe abstraite est une erreur, qu'il signale :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( ) = 0;
                ...
};

class Rectangle : public Figure
{
        int x1, y1, x2, y2;
        public:
                void seDessiner( )
                {
                        trait(x1, y1, x2, y1);
                        trait(x2, y1, x2, y2);
                        trait(x2, y2, x1, y2);
                        trait(x1, y2, x1, y1);
```

```
        }
        ...
};
```

Utilisation :

```
Figure f;           // ERREUR : Création d'une instance (objet) de la classe abstraite Figure
Figure *pt;         // Oui : c'est un pointeur
pt = new Figure;    // ERREUR : Création d'une instance (objet) de la classe abstraite Figure
Rectangle r;        // Oui : la classe Rectangle n'est pas abstraite
pt = new Rectangle; // Oui
```

4) Annexes

4.1) Classes abstraites

Une classe est ***abstraite*** lorsque l'une au moins de ses méthodes **virtuelles** est ***pure***. Lorsqu'une classe est abstraite, elle ne peut être utilisée directement ; en particulier, on ne peut pas déclarer d'objets de cette classe. Elle a pour vocation de posséder des sous classes par héritage. Elle sert à factoriser des données membres et des méthodes communes.

```
Class Personne
{
      public:
              // Méthode virtuelle pure
              virtual double calculerSalaire(Mois m)  = 0;
              ----------------------------------------
} ;

class Personnel : public Personne
{
      protected:
              unsigned int numero_bureau;
      public:
              // Constructeur
              Personnel(unsigned int numBureau, const char*name, const char*adress=0);
              // Accès au numéro de bureau
              unsigned int getNumeroBureau() const { return numero_bureau; }
              void setNumeroBureau(unsigned int n)  { numero_bureau = n; }
              // Redéfinition Méthode virtuelle pure
              double calculerSalaire(Mois m) {____________________};
};
```

Nous définissons une méthode **virtuelle *calculerSalaire*,** mais à ce niveau de la hiérarchie, nous ne pouvons pas donner de méthode de calcul du salaire : par défaut, nous avons renvoyé zéro. Il est possible de ne pas définir une méthode virtuelle dans une classe : elle devra alors obligatoirement être définie dans les classes dérivées (« classe Personnel »).

4.2) Le pointeur this

Rappel et résumé :

Une classe constitue une sorte de type, et les objets de cette classe en sont des ***instances***. La classe définit donc la structure des données, alors appelées *champs* ou *variables d'instances*, que les objets correspondants auront, ainsi que les méthodes de l'objet.

À chaque instanciation, une allocation de mémoire est faite pour les données du nouvel objet créé. L'initialisation de l'objet nouvellement créé est faite par une méthode spéciale, le *constructeur*. Lorsque l'objet est détruit, une autre méthode est appelée : le *destructeur*. L'utilisateur peut définir ses propres constructeurs et destructeurs d'objets si nécessaire.

Le pointeur this :

Chaque fois qu'un programme crée une **instance** d'une classe, le C++ crée un pointeur spécial appelé ***this*** contenant l'adresse de l'instance courante de la classe.

Le C++ connaît le pointeur ***this*** uniquement lorsqu'une méthode membre non-statique est en cours d'exécution.

L'utilisation du pointeur ***this*** est inutile mais beaucoup de programmeur exploitent ce fait pour améliorer la clarté.

```
class uneclasse
{
   public:
      uneclasse(char *titre, char *auteur);
      void afficher_avec_this( );
      void afficher_sans_this(uneclasse livre);

    private:
      char titre[256];
      char auteur[256];
};
```

```
/* *****************Programme Principal ************** */
void uneclasse::afficher_avec_this( )
{
   cout << "LIVRE: "<< this->titre <<endl;
   cout << "Auteur: "<< this->auteur <<endl;
}

void uneclasse::afficher_sans_this(uneclasse livre)
{
   cout << "LIVRE: "<< livre.titre <<endl;
   cout << "Auteur: "<< livre.auteur <<endl;
   cout << "LIVRE: "<< this->titre <<endl;
   cout << "Auteur: "<< this->auteur <<endl;
}

uneclasse::uneclasse(char *titre, char *auteur)
{
   strcpy(uneclasse::titre, titre);
   strcpy(uneclasse::auteur, auteur);
}


int main( )
{
   //Déclarations et initialisations d'objets de classe "uneclasse"
   uneclasse livre("La bible", "Jamsa");
    uneclasse livre1("La bible1", "Jamsa");
   uneclasse livre2("La bible2", "Jamsa");

   livre.afficher_avec_this( );
   livre.afficher_sans_this(livre2);

    cout <<"\n\n" ;
    system("PAUSE") ;
    return 0;
}
```

```
LIVRE: La bible
Auteur: Jamsa
LIVRE: La bible2
Auteur: Jamsa
LIVRE: La bible
Auteur: Jamsa

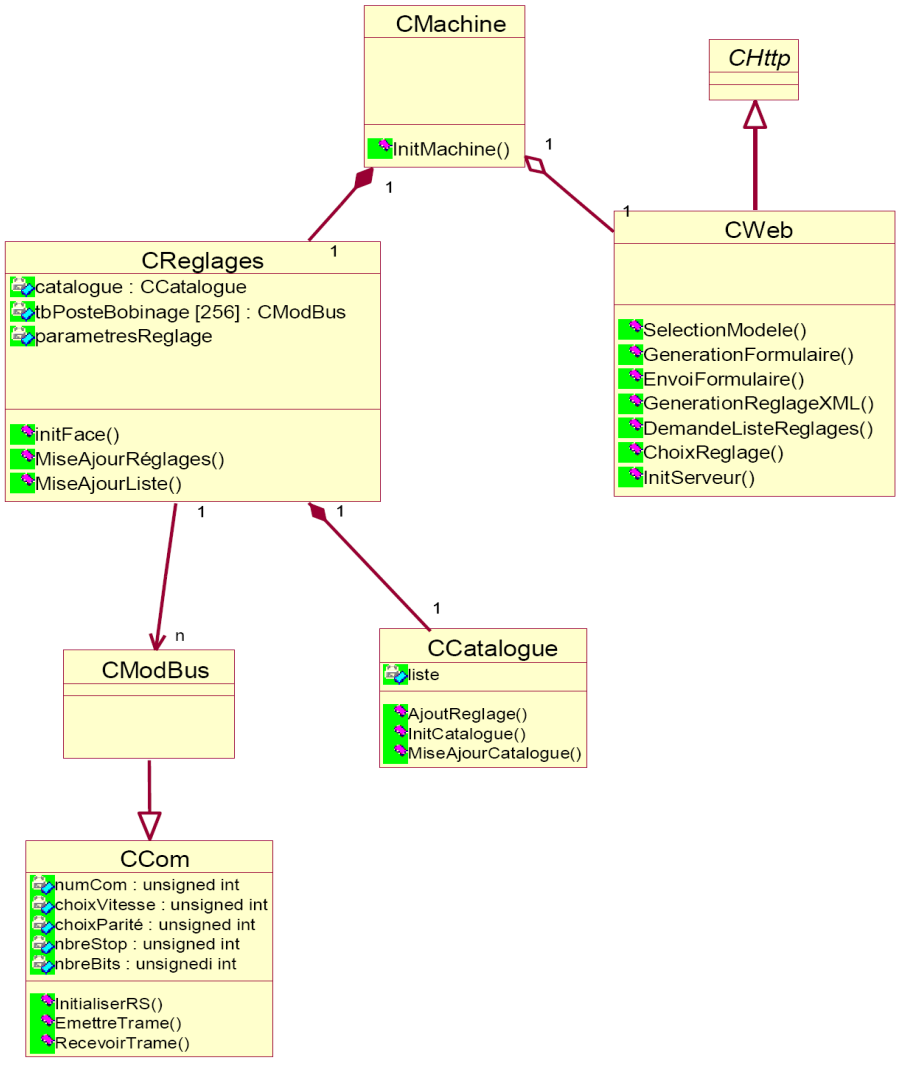
Appuyez sur une touche pour continuer...
```

Le pointeur ***this*** est le pointeur de l'objet courant ; il permet ici (voir exemple ci-dessus) que l'attribut et le paramètre puisse avoir le même nom.

4.3) Le diagramme de classe

Un diagramme de classe permet de décrire les interactions, les différentes relations entre différentes classes dans un modèle d'un logiciel.

a) Exemple de diagramme de classe UML (BTS IRIS 2006)



b) Schématisation de la relation entre les différentes classes

