Définition des variables en C++

- En C++ vous pouvez déclarer les variables ou fonctions n'importe où dans le code.
- La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.
- Ceci permet de définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité. C'est particulièrement utile pour des grosses fonctions ayant beaucoup de variables locales.

Exemple:

```
#include <stdio.h>

void main()
{
  int i = 0; // définition d'une variable
  i++; // instruction
  int j = 1; // définition d'une autre variable
  j++; // instruction
  int somme(int n1, int n2); // déclaration d'une fonction
  printf("%d + %d = %d\n", i, j, somme(i, j)); // instruction
}
```

Visibilité des variables

L'opérateur de résolution de portée **::** permet d'accéder aux variables globales plutôt qu'aux variables locales.

```
#include <iostream.h>

int i = 11;
void main()
{  int i = 34;
   {
     int i = 23;
     ::i = ::i + 1;
     cout << ::i << " " << i << endl;
   }
   cout << ::i << " " << i << endl;
}
//  le résultat de l'exécution –

//  12 23

//  12 34
```

L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (à cause de la lisibilité). Il est préférable de donner des noms différents aux variables plutôt que de réutiliser les mêmes noms.

Types composés

En C++, comme en langage C, le programmeur peut définir des nouveaux types en utilisant des ***struct***, ***enum*** ou ***union***. Mais contrairement au langage C, l'utilisation de ***typedef*** n'est plus obligatoire pour renommer un type.

Exemples:

```
struct FICHE    // définition du type FICHE
{
  char *nom, *prenom;
  int age;
};
```

// en C, il faut ajouter la ligne : **typedef struct** FICHE FICHE;

```
FICHE adherent, *liste;


enum BOOLEEN { FAUX, VRAI };
```

// en C, il faut ajouter la ligne : **typedef enum** BOOLEEN BOOLEEN;

```
BOOLEEN trouve;

trouve = FAUX;

trouve = 0; // ERREUR en C++ : vérification stricte des types
trouve = (BOOLEEN) 0; // OK
```

Définition d’une classe

La classe décrit le modèle structurel d'un objet composé de:

- ensemble des attributs (ou données membres) décrivant sa structure,
- ensemble des opérations (ou méthodes ou fonctions membres) qui lui sont applicables.

Une classe en C++ est une structure qui contient:

- des fonctions membres,
- des données membres.

Les mots réservés ***public***, ***private*** et ***protected*** délimitent les différentes sections des variables appartenant à la classe.

Exemple:

```
class Avion {
    public: // fonctions membres publiques
        void init(char [], char *, float);
        void affiche();
    private: // membres privées
        char immatriculation[6], *type; // données membres privées
        float poids;
        void erreur(char *message); // fonction membre privée
};      // n'oubliez pas ce ; après l'accolade !
```

Droits d'accès

L'**encapsulation** consiste à masquer l'accès à certains attributs et méthodes d'une classe. Elle est réalisée à l'aide des mots-clés:

- ***private*** **:**

  Les membres privés ne sont accessibles que par les fonctions membres de la classe. La partie privée est aussi appelée *réalisation.*

- ***protected*** **:**

  Les membres protégés sont comme les membres privés. Mais ils sont aussi accessibles par les fonctions membres des classes dérivées (*voir l'héritage*).

- ***public*** **:**

  Les membres publics sont accessibles par tous. La partie publique est aussi appelée *interface*.

Les mots réservés ***private***, ***protected*** et ***public*** peuvent figurer plusieurs fois dans la déclaration de la classe. Le droit d'accès ne change pas tant qu'un nouveau droit n'est pas spécifié.

Types de classes en C++

- **struct Classe1 { /* ... */ };**
  - tous les membres sont d'accès public par défaut,
  - le contrôle d'accès est modifiable,
  - cette structure est conservée pour pouvoir compiler des programmes écrits en C.

Exemple:

```
struct Date
{ // méthodes - publiques (par défaut)
      void set_date(int, int, int);
  private: // données privées
      int jour, mois, an;
};
```

- **union Classe2 { /* ... */ };**
  - tous les membres sont d'accès public par défaut,
  - le contrôle d'accès n'est pas modifiable.
- **class Classe3 { /* ... */ };**
  - tous les membres sont d'accès privé par défaut,
  - le contrôle d'accès est modifiable.

C'est ce dernier modèle de données qui est utilisée en programmation objet C++ pour définir des classes.

Recommandations de style

Mettre:

- la première lettre du nom de la classe est en majuscule,
- la liste des membres publics sont en premier,
- les noms des méthodes (des fonctions) sont en minuscules,
- le caractère _ est utilisé comme premier ou deuxième caractère du nom d'une donnée membre de la classe.

Définition des fonctions membres (1)

En général, la déclaration d'une classe contient seulement les définitions des données membres et les prototypes des fonctions membres.

```
class Avion
{
    public:
        void init(char [], char *, float);
        void affiche();
    private:
        char m_immatriculation[6], *m_type;
        float m_poids;
        void m_erreur(char *message);
};
```

Les fonctions membres sont définies dans un module séparé ou plus loin dans le code source.

Syntaxe d'une fonction appartenant à la classe Cls :

```
type_de_valeur_retournee Cls :: nom_de_fonction (paramètres_formels)
{ // corps de la fonction }
```

Définition des fonctions membres (2)

La définition de la *méthode* (la fonction membre) peut avoir lieu à l'intérieur de la déclaration de la classe. Dans ce cas, ces fonctions sont automatiquement traitées par le compilateur comme des fonctions ***inline***.

Une fonction membre définie à extérieur de la classe peut être aussi qualifiée explicitement de fonction ***inline***.

Exemple:

```
class Nombre
{
        public:
          void setnbre (int n) { nbre = n; } // fonction inline
          int getter () { return nbre; } // fonction inline
          void affiche ();
        private:
          int nbre;
};

inline void Nombre :: affiche()     // fonction inline
{    cout << "Nombre = " << nbre << endl;    }
```

Instanciation d'une classe

De façon similaire à une *struct* ou à une *union*, le nom de la classe représente un nouveau type de données.

On peut donc définir des variables de ce nouveau type; on dit alors que l'on déclare des **objets** ou des **instances** de cette classe.

Exemples:

```
Avion av1; // une instance simple de la classe Avion (statique)

Avion *av2; // un pointeur (non initialisé) sur un objet de classe Avion

Avion compagnie[10]; // un tableau d'instances de la classe Avion

av2 = new Avion; // création (dynamique) d'une instance d'Avion
```

Utilisation des objets d’une classe

Après avoir créé une instance (de façon statique ou dynamique) on peut accéder aux données et méthodes de la classe.

Cet accès se fait comme pour les structures à l'aide des opérateurs:

. (point) et **-> (flèche)**.

Exemples:

```
Avion av1; // une instance simple de la classe Avion

Avion *av2 = new Avion; // initialisation d’un pointeur

Avion compagnie[10]; // un tableau d'instances

// exemples des appels des méthodes de la classe
av1.init("FABCD", "TB20", 1.47); // appel de la méthode init

av2->init("FCDEF", "ATR 42", 80.0); // appel de la méthode init
compagnie[0].init("FEFGH","A320", 150.0); // appel de la méthode init

av1.affiche(); // appel de la méthode affiche

av2->affiche();// appel de la méthode affiche

compagnie[0].affiche();// appel de la méthode affiche

av1.m_poids = 0; // ! erreur, si m_poids est un membre privé !
```

Fonctions membres constantes (1)

Certaines méthodes d'une classe ne doivent (ou ne peuvent) pas modifier les valeurs des données membres de la classe, ni retourner une référence non constante ou un pointeur non constant d'une donnée membre : on dit que ce sont des fonctions membres constantes. Ce type de déclaration renforce les contrôles effectués par le compilateur et permet donc une programmation plus sûre. Il est donc *très souhaitable* d'en déclarer aussi souvent que possible dans les classes.

Exemple:

```
class Nombre {
                    public :
                       void setnbre(int n) { nbre = n; }
                      // méthodes constantes
                      int getnbre() const { return nbre; }
                      void affiche() const;
                    private :
                      int nbre;

                  };
inline void Nombre :: affiche() const

      {    cout << "Nombre = " << nbre << endl;    }
```

Fonctions membres constantes (2)

Une fonction membre *const* peut être appelée sur des objets constants ou pas, alors qu'une fonction membre non constante ne peut être appelée que sur des objets non constants.

Exemples:

```
// voir la déclaration de la classe Nombre

    const Nombre n1; // un objet constant de la classe Nombre
    n1.affiche(); // OK
    n1.setnbre(15); // ERREUR: seules les fonctions const peuvent être
    //appelées pour un objet constant


    Nombre n2;
    n2.affiche(); // OK
    n2.setnbre(15); // OK
```

Constructeurs et destructeurs

- Les données membres d'une classe ne peuvent pas être initialisées durant leur définition. Il faut donc prévoir une méthode d'initialisation de celles-ci.

- De même, après avoir fini d'utiliser un objet, il est bon de prévoir une méthode permettant de détruire l'objet (libération de la mémoire dynamique ...).

Le **constructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à l'instanciation de l'objet, assurant ainsi une initialisation correcte de l'objet. Le constructeur est une fonction qui porte comme nom, le nom de la classe et qui ne retourne pas de valeur (pas même un *void*).

De la même façon que pour le constructeur, le **destructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet.

Constructeur par défaut

On appelle **constructeur par défaut** un constructeur n'ayant pas de paramètres ou ayant des valeurs par défaut pour tous les paramètres.

```
class Nombre
{
    public:
        Nombre (int i = 0); // constructeur par défaut à 1 paramètre
        Nombre (int i = 1, int j = 2); // constructeur par défaut à 2 paramètres
        Nombre (int i, int j, int k); // constructeur à 3 paramètres
};
```

Si le concepteur de la classe ne spécifie pas de constructeur, le compilateur générera un constructeur par défaut. Comme les autres fonctions, les constructeurs peuvent être surchargés. Le constructeur est appelé à l'instanciation de l'objet. Il n'est pas appelé quand on définit un pointeur sur un objet.

Exemples d’utilisation d’un constructeur

Nombre n1; // correct, appel du constructeur par défaut

Nombre n2(10); // correct, appel du constructeur à 1 paramètre

Nombre *ptr1, *ptr2; // correct, pas d'appel aux constructeurs

ptr1 = **new** Nombre; // appel du constructeur par défaut

ptr2 = **new** Nombre(12); // appel du constructeur à 1 paramètre

Nombre tab1[10]; // chaque objet du tableau est initialisé par

// un appel au constructeur par défaut

Nombre tab2[3] = { Nombre(10), Nombre(20), Nombre(30) };

// initialisation des 3 objets du tableau par les nombres 10, 20 et 30

Destructeur d’une classe

Le destructeur est une fonction:

- qui porte comme nom, le nom de la classe précédé du caractère ~ (tilde),
- qui ne retourne pas de valeur (pas même un *void*),
- qui n'accepte aucun paramètre (le destructeur ne peut donc pas être surchargé).

Exemple:

```
class Exemple {
                    public:
                            ~Exemple();
                  };
Exemple :: ~Exemple() { delete (…) }
```

Comme pour le constructeur, le compilateur générera un destructeur par défaut si le concepteur de la classe n'en spécifie pas un.

Membres statiques d’une classe

Ces membres sont utiles lorsque l'on a besoin de gérer des données communes aux instances d'une même classe.

Données membres statiques:

Si l'on déclare une donnée membre comme **static**, elle aura la même valeur pour toutes les instances de cette classe.

```
class Ex1  {
        public: Ex1() { nb++; /* ... */ }
                ~Ex1() { nb--; /* ... */ }
        private:
                static int nb; // initialisation impossible ici
    };
```

L'initialisation de cette donnée membre statique se fera en dehors de la classe et en global par une déclaration suivante:

```
int Ex1 :: nb = 0;          // initialisation du membre statique nb
```

Fonctions membres statiques d’une classe

De même que les données membres statiques, il existe des fonctions membres statiques. Ces fonctions:

- ne peuvent accéder qu'aux membres statiques,
- ne peuvent pas être surchargées,
- existent même s'il n'y a pas d'instance de la classe.

```
class Ex1 {
  public:
    static void affiche() { cout << nb << endl;    } // fonction membre statique
  private:
    static int nb;
};
int Ex1 :: nb = 0;          // initialisation du membre static (en global)
void main()
{ Ex1 :: affiche();
  Ex1 a, b, c;
  a.affiche();
}
```

Modèle de la classe queue (1)

```
#include <iostream.h>
class queue          //on déclare la classe queue

{
            int q[10];
            int sloc, rloc;
   public:
            void init (void);
            void qput (int);
            int qget (void);
};

// la description des fonctions membres de la classe queue
void queue :: init (void)
{
          rloc = sloc = 0;
}

int queue :: qget (void)
{
          if (sloc == rloc) {
                cout << " La queue est vide ";
                return 0; // il faut retourner une valeur
          }
          return q[rloc++];
}

void queue :: qput (int i)
{
          if (sloc == 10) {
                cout << " La queue est pleine ";
                return;
          }
          q[sloc++] = i;
}
```

Modèle de la classe queue (2)

```
//la fonction principale main

main (void)
{
        queue a, b;// la création des objets a et b de la classe queue

        a.init();      // l'initialisation de la queue a
        b.init();      // l'initialisation de la queue b

        a.qput(7);  // le remplissage de la queue a
        a.qput(9);
        a.qput(11);

        cout << a.qget() << " ";
        cout << a.qget() << " ";
        cout << a.qget() << " ";
        cout << a.qget() << "\n";

// on peut remplacer les quatre derniers opérateurs par un seul
//cout << a.qget() << " " << a.qget() << " " << a.qget() << " " << a.qget() << "\n";

        for (int i = 0; i < 12; i++)       // le remplissage de la queue b par i au carré
              b.qput(i*i);

        for (i = 0; i < 12; i++)
              cout << b.qget() << " ";
        cout << " \n \n ";

        return 0;
}
```

Utilisation de constructeurs et de destructeurs (1)

```
#include <iostream.h>

//on déclare la classe queue
class queue
{
        int q[10];
        int sloc, rloc;
  public:
        queue (void);          // le constructeur
        ~queue (void);         // le destructeur
        void qput (int);
        int qget (void);
};

// la description du constructeur de la classe queue
queue :: queue (void)
{
        rloc = sloc = 0;
        cout << " La queue est initialisée \n " ;
}

// la description du destructeur de la classe queue
queue :: ~queue (void)
{
        cout << " La queue est détruite \n " ;
}

void queue :: qput (int i)
{
        if (sloc == 10) {
                cout << " La queue est pleine ";
                return;
        }
        q[sloc++] = i;
}
```

Utilisation de constructeurs et de destructeurs (2)

```
int queue :: qget (void)
{
        if (rloc == sloc) {
            cout << " La queue est vide ";
            return 0;
        }
        return q[rloc++];
}

//la fonction principale main
main (void)
{
        queue a, b;         // la création des objets a et b de la classe queue

        a.qput(10);
        b.qput(19);
        a.qput(20);
        b.qput(1);

        cout << a.qget() << " ";
        cout << a.qget() << " ";
        cout << b.qget() << " ";
        cout << b.qget() << "\n";
        return 0;
}
// Ce programme affiche comme résultat :
// La queue est initialisée
// La queue est initialisée
10 20 19 1
// La queue est détruite
// La queue est détruite
```