

Dépasser ses programmes

1. Oublier les réflexes du langage C

Les programmeurs habitués au langage C ont sans doute prévu de nombreuses macros instructions traitées par le préprocesseur. C'est notamment le cas des constantes textuelles et de "fonctions" qui rendent bien des services.

Le langage C++ propose cependant des alternatives qui améliorent la sûreté du programme ainsi que sa rapidité.

Les constantes définies avec le mot-clé `const`, tout d'abord, présentent l'énorme avantage d'être vérifiées par le compilateur et non par le préprocesseur. En cas d'erreur - visibilité, orthographe... - le compilateur peut donner beaucoup plus de contexte au programmeur.

Les fonctions en ligne (`inline`) permettent de développer des instructions sans construire tout un cadre de pile, avec les conséquences que cela peut avoir sur la transmission des arguments et sur les performances. Naturellement, leur mise en œuvre doit satisfaire à quelques contraintes mais il y a des bénéfices certains à les employer.

Considérons l'exemple suivant, dans lequel une constante symbolique est définie à l'attention du préprocesseur. En cas d'erreur d'écriture dans le `main`, le compilateur n'a aucun moyen de préciser dans quel fichier `.h` elle pourrait ne pas correspondre. De même, nous avons décrit au chapitre De C à C++ une macro qui compare deux entiers. Mais s'agissant d'une macro qui répète l'un de ses arguments, nous constatons que des effets de bord apparaissent.

L'exemple suivant propose une alternative à chaque problème, basée sur la syntaxe C++ plus sûre et plus rapide :

```
#define TAILLE 10
#define COMP(a,b) (a-b>0?1:(a-b==0?0:-1))

const int taille = 10;
inline int comp(int a,int b)
```

```

{
    if(a-b>0)
        return 1;
    else
    {
        if(a==b)
            return 0;
        else
            return -1;
    }
}

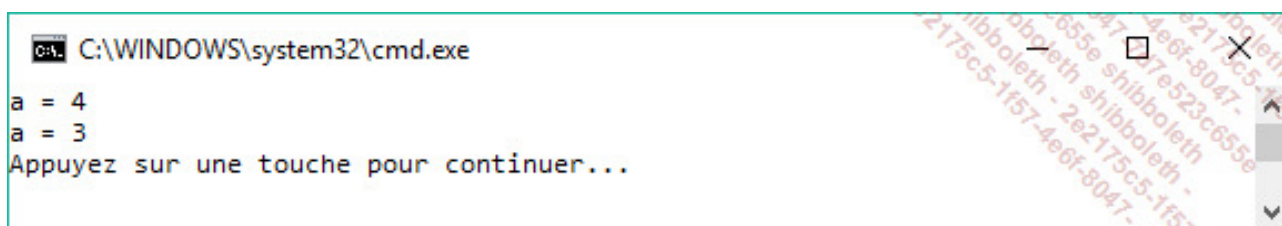
int
{
    // utilisation comme en langage C
    int* tableau = new int[TAILLE];
    int a = 2, b = 3;
    int z = COMP(a++, b); // quelle est la valeur de a en sortie ?
    printf("a = %d\n",a);

    // utilisation comme en C++
    A = 2;
    Z = comp(a++,b); // quelle est la valeur de a en sortie ?
    printf("a = %d\n",a);

    return 0;
}

```

À l'exécution, nous vérifions que l'utilisation de la macro produit un effet de bord indésirable sur la variable `a` :



```

C:\WINDOWS\system32\cmd.exe
a = 4
a = 3
Appuyez sur une touche pour continuer...

```

2. Gestion de la mémoire

L'une des subtilités dont nous n'avons pas encore parlé est l'allocation puis la libération de tableaux d'objets. Considérons la classe `Point` dont le constructeur alloue de la mémoire pour représenter des coordonnées :

```
class Point
{
public:
    int* coordonnees;

    Point()
    {
        coordonnees = new int[2];
        printf("Point\n");
    }

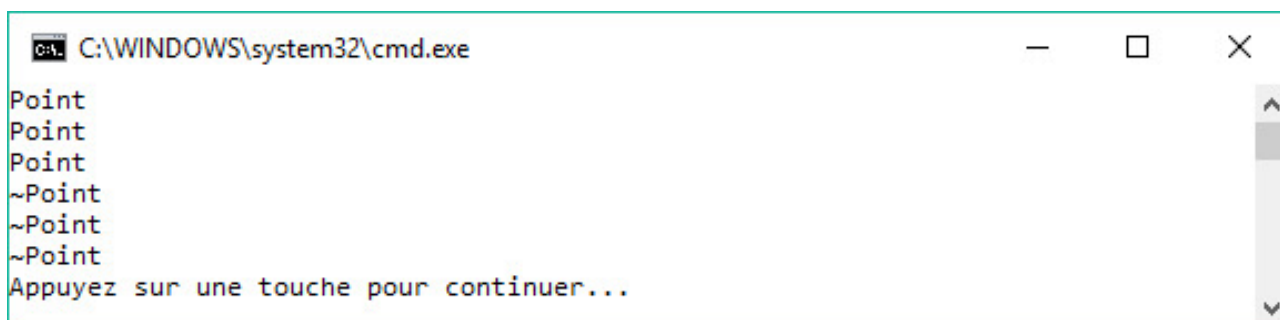
    ~Point()
    {
        delete coordonnees;
        printf("~Point\n");
    }
};
```

Si nous procédons à l'instanciation d'un tableau de `Point`, il y aura autant d'appels au constructeur que d'objets créés :

```
Point* tab = new Point[3]; // trois constructeurs appelés car
3 points créés
```

Après utilisation, il est convenable de libérer ce tableau. Toutefois, l'utilisation de `delete` sur cet objet `tab` peut avoir des comportements inattendus. Une partie des objets ne sera pas détruite, ou pire, une erreur d'exécution peut survenir. Il convient plutôt d'utiliser l'instruction `delete[]` :

```
//delete tab; // erreur, seul un destructeur est appelé
delete[] tab; // erreur, seul un destructeur est appelé
```



3. Concevoir des classes avec soin

Sans vouloir appauvrir les nombreuses normes élaborées pour C++, nous recommandons d'adopter un schéma type pour définir des classes en se concentrant sur le programme, mais sans oublier l'essentiel. Ce langage est à la fois très puissant, très riche, et recèle beaucoup de pièges dans ses constructions "non dites".

La première chose à faire est de choisir des identificateurs qui sont explicites et parlants, et en même temps de respecter une convention de nommage. Mélanger `m_param`, `iParam`, `param`, tout appeler `champ1`, `champ2`, `champ3` sont des erreurs à éviter à tout prix.

Une classe est une entité complète, autonome, et le programmeur doit connaître à l'avance les membres qui la composent.

La seconde préoccupation est la visibilité de ses membres ; oublions les formules toutes faites "tous les champs sont privés et toutes les méthodes sont publiques". La réalité est toujours plus nuancée, et les méthodes d'accès - pour ne pas dire les propriétés - viennent rapidement contredire cette logique un peu simpliste.

La troisième recommandation est l'ordre dans lequel sont définis les membres. On peut commencer par les champs, puis les constructeurs, puis les méthodes publiques, enfin les méthodes non publiques. Il ne faut pas se priver de répéter pour chaque membre sa visibilité, ce qui évite des erreurs en cas de copier-coller intempestif.

Pour chaque méthode, on doit bien avoir à l'esprit si elle est statique, virtuelle, abstraite... Pour chaque champ, il faut déterminer son type, son nom, sa visibilité, s'il est statique, constant...

La dernière recommandation est la documentation. Une bonne classe doit être accompagnée de beaucoup de commentaires. Ceux-ci ne doivent pas reformuler en moins bien ce qui est écrit en C++, mais plutôt développer des détails supplémentaires pour guider celui qui parcourt le code.

Comme C++ sépare la définition de l'implémentation, cela constitue une "optimisation" pour le programmeur. Le compilateur ne "moulinera" que les fichiers `.cpp` qui sont amenés à évoluer plus fréquemment que les fichiers `.h`, normalement plus stables car issus de l'étape de conception.

4. Y voir plus clair parmi les possibilités de l'héritage

Le langage C++ propose plusieurs modes d'héritage. Le plus utilisé, à juste titre, est l'héritage public. À quoi peuvent bien servir les autres modes, comme l'héritage privé ?

L'héritage public spécialise une classe. La classe de base exprime le concept le plus général, le plus abstrait, alors que les sous-classes vont plus dans le détail. Mais il existe toujours en principe une relation "est-un-cas-particulier-de" entre la classe dérivée et la classe de base ; le livret est un cas particulier de compte. La voiture est un cas particulier de véhicule...

De ce fait, il faut se méfier des constructions syntaxiquement possibles mais sémantiquement insensées. La classe `Chateau-Satellite` n'a sans doute pas grande utilité. Une applet ne peut être aussi une horloge et un gestionnaire de souris. Ces constructions amènent de la complexité et produisent des programmes difficiles à maintenir. Faut-il en conclure que l'héritage multiple est à proscrire ? Sans doute pas. C'est un mécanisme indispensable mais qui doit être envisagé avec des précautions. C++ étant un langage général, il permet l'héritage multiple, il en a même eu besoin pour la STL. Mais le programmeur doit être prudent s'il est amené à recourir lui-même à cette construction. L'autre possibilité est de s'appuyer sur un framework qui prévoit l'héritage multiple dans un contexte encadré, sans risque.

Et l'héritage privé, finalement ? Là encore il s'agit d'une "astuce" pour éviter l'explosion de code au sein d'un programme complexe. Tous les membres devenant privés, ce n'est pas pour créer des méthodes virtuelles que l'on recourt à ce mode. C'est pour réutiliser du code sans trop s'embarrasser d'un quelconque polymorphisme.

De nos jours, il existe d'autres façons de contrôler la quantité de code, et la programmation par aspects AOP (*Aspect-Oriented Programming*) ouvre une voie prometteuse.

5. Analyser l'exécution d'un programme C++

Il existe des outils d'analyse de logiciels C++ (*profilers*) traquant les goulets de performances, les fuites mémoire et les opérations litigieuses. Sous Unix, l'utilitaire Purify sert précisément à détecter les erreurs d'implémentations d'un programme C++.

Comme C++ est un langage assez proche de la machine, on peut également employer sous Windows l'utilitaire Process Viewer pour analyser le fonctionnement d'un programme.