Organisation des programmes

1. Espaces de noms

Le langage C ne connaît que deux niveaux de portée : le niveau global, auquel la fonction `main()` appartient, et le niveau local, destiné aux instructions et aux variables locales. Avec l'apparition de classes, un niveau supplémentaire s'est installé, celui destiné à l'enregistrement des champs et des méthodes. Puis l'introduction de la dérivation (héritage) et des membres statiques a encore nuancé la palette des niveaux de portée.

Pour les raisons évoquées en introduction, il devenait nécessaire de structurer l'espace global. Pour n'en retenir qu'une, l'espace global est trop risqué pour le rangement de variables et de fonctions provenant de programmes anciens. Les conflits sont inévitables.

On peut alors partitionner cet espace global à l'aide d'espaces de noms :

```
namespace Batiment
{
 double longueur;

 void mesurer()
 {
  longueur=50.3;
 }
};

namespace Chaines
{
 int longueur;

 void calcule_longueur(char*s)
 {
  longueur=strlen(s);
 }
};
```

Deux espaces de noms, `Batiment` et `Chaines`, contiennent tous les deux une variable

nommée `longueur`, d'ailleurs de type différent. Les fonctions `mesurer()` et `calcule_longueur()` utilisent toujours la bonne version, car la règle d'accessibilité est également vérifiée dans les espaces de noms : le compilateur cherche toujours la version la plus proche.

Pour utiliser l'une ou l'autre de ces fonctions, la fonction `main()` doit recourir à l'opération de résolution de portée `::` ou bien à une instruction **`using`** :

```
int main(int argc, char* argv[])
{
  Batiment::mesurer();
  printf("La longueur du bâtiment est %g\n",Batiment::longueur);

  using Chaines::longueur;
  Chaines::calcule_longueur("bonjour");
  printf("La longueur de la chaîne est %d\n",longueur);
  return 0;
}
```

Nous remarquons que l'appel d'une fonction déclarée à l'intérieur d'un espace de noms ressemble beaucoup à celui d'une méthode statique. Cette analogie se prolonge pour l'accès à une variable, que l'on peut comparer à l'accès à un champ statique.

La syntaxe `using` est utile pour créer des alias. Avant la déclaration `using Chaines::longueur`, il n'existe pour `main()` aucune variable accessible. Ensuite, jusqu'à nouvel ordre, `longueur` est devenu un alias de `Chaines::longueur`.

a. Utilisation complète d'un espace de noms

Il n'est pas rare d'avoir à utiliser tous les membres appartenant à un espace de noms. Il est d'autant plus inutile de créer un alias pour chacun d'eux que l'opération risque d'être à la fois fastidieuse et vaine. La syntaxe **`using namespace`** convient bien mieux :

```
#include <iostream>
using namespace std;
// utilise tout l'espace de noms std
```

Cette déclaration est forte de signification, et il convient de bien la différencier de la directive `#include`. Alors que la directive `#include` inclut le fichier indiqué, `iostream` en l'espèce, la directive `using namespace` crée des alias pour les membres de l'espace de noms `std`, déclarés dans `iostream`.

Autrement dit, la directive `using` n'importe pas, n'inclut pas, elle est uniquement destinée à simplifier l'écriture. Sans son emploi, nous devrions préfixer chaque élément par `std`.

Ainsi est-il plus commode d'écrire :

```
cout << "Bonjour\n";
```

que :

```
std::cout << "Bonjour\n";
```

Toutefois, en cas de conflit entre deux espaces de noms entièrement utilisés, l'utilisation de l'opérateur de résolution de portée lève toute ambiguïté :

```
namespace Chaines
{
 int longueur;

 void calcule_longueur(char*s)
 {
  longueur=strlen(s);
 }

 bool cout;
};
...
 using namespace Chaines;
 std::cout << "Pas de doute sur cette ligne\n";
```

b. Espace de noms réparti sur plusieurs fichiers

Il est tout à fait juste de définir le même espace de noms au travers de différents fichiers de code source, chacun d'eux contribuant à l'enrichir de ses membres.

```
// tri.h

namespace Algo
{
  void tri_rapide(int*valeurs);
  void tri_bulle(int*valeurs);
};

// pile.h

namespace Algo
{
  class Pile
  {
  protected:
    int*valeurs;
    int nb_elements;
    int max;
  public:
    Pile();
    void empiler(int v);
    int depiler();
    bool pile_vide();
  };
};
...
#include "tri.h"
#include "pile.h"

using namespace Algo;

int main(int argc, char* argv[])
{
    return 0;
}
```

Le compilateur C++ de Microsoft, lorsqu'il fonctionne en mode managé (avec les

extensions .NET), conserve l'organisation des modules au niveau du code objet et de l'exécutable. Du même coup, la différence entre la directive `#include` et `using namespace` devient de plus en plus ténue ; le langage C# a donc fait le choix de réunir les deux directives en une seule, `using`, un peu à la manière de Java avec l'instruction `import`.

D'autre part, il est tout à fait possible de conserver la séparation entre déclaration et définition. L'organisation du code source en `.h` et en `.cpp` fonctionne ainsi comme à l'accoutumée :

```
// tri.cpp
#include "tri.h"

void Algo::tri_rapide(int*valeurs)
{

}
```

c. Relation entre classe et espace de noms

Finalement, classe et espace de noms sont assez proches. Doit-on considérer l'espace de noms comme une classe ne contenant que des membres statiques ou la classe comme un module instanciable ? En réalité, une classe C++ engendre son propre espace de noms. Voilà qui éclaire à la fois l'utilisation de l'opérateur de résolution de portée et surtout, qui explique la déclaration des champs statiques.

Si le champ statique était effectivement "instancié" au moment de sa déclaration dans le corps de la classe, ce champ pourrait exister en plusieurs versions : chaque fichier `.cpp` incluant la déclaration de la classe au travers d'un fichier d'en-tête `.h` provoquant sa duplication. Il faut donc instancier ce champ, une et une seule fois, dans un fichier `.cpp`, normalement celui qui porte la définition des méthodes de la classe.

La mise en garde vaut aussi pour les espaces de noms. Si un espace est défini dans un fichier `.h`, il faut veiller à ne pas déclarer de variable dans cette partie, autrement la variable pourrait se trouver définie à plusieurs endroits.

Comment donc indiquer au lecteur l'existence d'une variable dans la déclaration `.h` d'un

espace de noms, telle `cout` et `cin` pour l'espace `std`, sachant qu'elle ne peut pas être définie à cet endroit ? La réponse est encore le mot-clé `static`. Le mot-clé `static` prévient le compilateur que cette variable ne doit pas être dupliquée par chaque fichier incluant la définition de l'espace de noms.

Toutefois, et contrairement aux classes, la variable n'a pas à être définie dans un fichier `.cpp` :

```
// pile.h

namespace Algo
{
  class Pile
  {
  protected:
    int*valeurs;
   int nb_elements;
    int max;
  public:
    Pile();
    void empiler(int v);
    int depiler();
    bool pile_vide();
  };

  static int T_PILE=3;
};

// pile.cpp
#include "pile.h"

// ne pas déclarer ici la variable Algo::T_PILE
```

d. Déclaration de sous-espaces de noms

Un espace de noms peut très bien contenir d'autres espaces de noms, voire des classes et des structures.

```
namespace Algo
{
 namespace Structures
 {
  class Tableau {};
  class Pile {};
 };

 namespace Fonctions
 {
  using namespace Structures;
  void tri_rapide(Tableau t);
 };
};
```

Il est absolument nécessaire d'utiliser l'espace `Structures` dans l'espace `Fonctions` pour atteindre la définition de la classe `Tableau`. À l'extérieur de l'espace `Algo`, plusieurs directives peuvent être nécessaires pour utiliser l'ensemble des membres déclarés dans `Algo` :

```
using namespace Algo;
using namespace Algo::Structures;

int main(int argc, char* argv[])
{
 Tableau t;
 return 0;
}
```

2. Présentation de la STL

La bibliothèque standard a été créée dans le but d'aider le programmeur C++ à produire des logiciels de haut niveau. Bien que les langages C et C++ soient universellement supportés par des interfaces de programmation (API) de toutes sortes, la partie algorithmique reste un peu en retrait tant les instructions sont concises et proches de la

machine. Les chaînes de caractères ASCII-Z (terminées par un octet de valeur zéro) sont un bon exemple de situation pour laquelle le programmeur se trouve démuni. De nos jours, la question n'est plus de savoir comment les chaînes sont représentées mais plutôt comment les utiliser le mieux possible. Il faut conserver de bonnes performances, certes, mais il faut aussi s'accommoder des codages internationaux.

Aussi, la généralisation de la programmation orientée objet a fini par dénaturer le travail des développeurs. L'algorithmie a été abandonnée au profit d'une vision abstraite, à base d'interfaces. Bjarne Stroustrup avait peut-être senti que cette transformation du métier de programmeur interviendrait rapidement, aussi la STL fut-elle rendue disponible très vite après la publication du langage.

La STL propose un certain nombre de thèmes pour aider le programmeur : les chaînes de caractères, les entrées-sorties, les algorithmes et leurs structures de données, le calcul numérique. Si cela ne suffisait pas à vos travaux, rappelez-vous qu'elle a été bâtie en C++, aussi tout programmeur est-il libre de la compléter avec ses propres contributions.