MM031 - TP3

Maxime Chupin : chupin@ann.jussieu.fr

26 Janvier 2016

1 Rappels

Déclaration d'une fonction

```
type_retour nom_de_la_fonction(type arg1, type arg2, ....){
  /*liste d'instructions*/
}
```

Appel d'une fonction

```
nom_de_la_fonction(arg1, arg2, ...);
```

Exemple

Voici un exemple de fichier source : fonction.cpp

```
#include <iostream>

using namespace std;

int max(int a, int b) { if (a>b) return a; else return b; }

int main() {
  int i = 5, j = 1;
  cout << "max(" << i << ", " << j << ") = " << max(i,j) << endl;
  return 0;
}
```

Pointeurs

Déclaration d'un pointeur

```
type* nom_pointeur ;
```

Allocation mémoire d'un pointeur

```
nom_pointeur = new type ;
```

Désallocation mémoire d'un pointeur

```
delete nom_pointeur;
```

Allocation d'un tableau dynamique

```
type* nom_pointeur = new type[taille];
```

Désallocation d'un tableau dynamique

```
delete [] nom\_pointeur;
```

2 Compilation

La compilation consiste en une série d'étapes de transformation du code source en du code machine exécutable sur un processeur cible.

2.1 Phases de la compilation

La compilation passe par différentes phases, produisant ou non des fichiers intermédiaires :

→ *préprocessing* : Le code source original est transformé en code source brut. Les commentaires sont enlevés et les directives de compilation commençant par # sont d'abord traités pour obtenir le code source brut ;

→ *compilation en fichier objet* : les fichiers de code source brut sont transformés en un fichier dit objet, c'est-à-dire un fichier contenant du code machine ainsi que toutes les informations nécessaires pour l'étape suivante (édition des liens). Généralement, ces fichiers portent l'extension `.obj` ou `.o` ;

→ *édition de liens* : dans cette phase, l'éditeur de liens (linker) s'occupe d'assembler les fichiers objet en une entité exécutable et doit pour ce faire résoudre toutes les adresses non encore résolues, tant des mémoires adressées que des appels de fonction. L'entité exécutable est généralement soit un exécutable, soit une bibliothèque dynamique (DLLs sous Windows et toutes les variantes, tels que objet COM, OCX, etc, et les `.so` sous Linux). Les compilateurs sont capables de générer des bibliothèques statiques, qui sont en quelques sortes le rassemblement d'un ensemble de fichiers objet au sein d'un même fichier. Dans ce cas, la phase d'édition de liens n'a pas eu lieu.

Cette découpe en phases permet de compiler séparément les bibliothèques en fichiers objets, et l'application et évite donc de tout re-compiler, ce qui prendrait beaucoup de temps pour les applications ayant un code source important.

2.2 Construire un `Makefile`

Les `Makefiles` sont des fichiers, généralement appelés `makefile` ou `Makefile`, utilisés par le programme `make` pour exécuter un ensemble d'actions, comme la compilation d'un projet, l'archivage de document, la mise à jour de site, etc.

Pour la programmation en C++, il s'agit d'automatiser les compilations des fichiers séparés pour n'avoir, grâce à une commande *unique* (`make`), qu'à (re)compiler que les fichiers nécessaires.

Pour des introductions à l'écriture de `Makefile`, voici quelques liens utiles. À partir de maintenant, vous devriez écrire un `Makefile` pour tous vos codes.

→ http://gl.developpez.com/tutoriel/outil/makefile
→ http://www.gnu.org/software/make/manual/

Exemple de *Makefile*

```
CC=g++
CCFLAGS = -Wall -g
LDFLAGS =
LIBRARY =
INCLUDES =

EXEC = main
SRCS = main.cpp
OBJS = main.o

%.o:%.cpp
    $(CC) $(CCFLAGS) $(INCLUDES) -c $<

all: $(EXEC)

$(EXEC): $(OBJS)
    $(CC) $(CCFLAGS) $(OBJS) $(LIBRARY) $(LDFLAGS) -o  $@

clean: rm -f $(OBJS) $(EXEC)
```

3 Classes

Déclaration d'une classe

Fichier header (.hpp)

```
class nom_de_la_classe{
   private:
     /* listes des champs privés */
   public:
     /* listes des membres public */
};
```

Le fichier .cpp correspondant contient alors les définitions des différentes méthodes, les constructeurs, etc. propres à la (les) classe(s) créée(s).

3.1 Fonction membre ou fonction amie

Un des buts de ce travail est de vous faire réfléchir au rôle d'une fonction agissant sur une classe que vous avez définie. Est-ce qu'il doit s'agir d'une fonction membre, ou d'une fonction amie. Il peut bien sûr y avoir des choix différents et pertinents, mais il faut s'être posé la question.

Exercice 1 : vecteur 2D

1. Définir une classe vecteur constituée de deux `double` permettant de stocker les coordonnées (x, y) d'un vecteur.
2. Définir un constructeur qui par défaut donne la valeur 0 aux deux coordonnées.

3. Définir un constructeur par recopie.
4. Définir une fonction d'affichage pour la classe `vecteur`.
5. Définir une fonction renvoyant un `double` contenant la norme d'un objet `vecteur`.
6. Définir deux fonctions renvoyant un `double` contenant respectivement le produit scalaire et le determinant de deux objets `vecteur`.
7. Définir deux fonctions renvoyant un objet `vecteur` contenant respectivement la somme, le produit coordonnée à coordonnée de deux objets `vecteur`.
8. Tester toutes ces fonctions dans un programme principal.

Exercice 2 : matrice 2×2

1. Définir une classe `matrice` 2×2 constituée de quatre `double`s.
2. Définir un constructeur qui par défaut donne la valeur 0 aux quatre éléments de la matrice.
3. Définir un constructeur par recopie.
4. Définir une fonction d'affichage pour la classe `matrice`.
5. Définir une fonction renvoyant un `double` contenant le déterminant d'un objet `matrice`.
6. Définir deux fonctions renvoyant un objet `matrice` contenant respectivement la matrice transposée et la comatrice d'un objet `matrice`.
   Rappels :
$$\text{Si } A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$
$$\text{sa comatrice est } \mathrm{Com}(A) = \begin{bmatrix} d & -c \\ -b & a \end{bmatrix}$$
7. Définir une fonction renvoyant un objet `matrice` contenant le produit d'un `double` par un objet `matrice`.
8. Définir une fonction renvoyant un objet `matrice` contenant la matrice inverse d'un objet `matrice`. On pourra utiliser la formule (valable pour des matrices 2×2) :
$$A^{-1} = \frac{1}{\det(A)} \mathrm{Com}(A)^T, \quad T \text{ désignant la transposition.}$$
9. Définir une fonction renvoyant un objet `vecteur` contenant le produit d'un objet `vecteur` et d'un objet `matrice`.
10. Définir une fonction renvoyant un objet `matrice` contenant le produit de deux objets `matrice`.
11. Tester toutes ces fonctions dans le programme principal.

Exercice 3 : Champ de vecteurs 2D

1. Définir une classe `champ` constituée d'un entier `nb_vect` et d'un pointeur sur objet de type `vecteur`.

```
class champ
{
private :
  int nb_vect;
  vecteur* p;
};
```

2. Définir un constructeur par défault pour la classe `champ`.

3. Définir un constructeur qui permet à l'utilisateur de rentrer un tableau d'objets de type `vecteur` et sa taille.
4. Définir un destructeur pour la classe `champ`.
5. Définir un constructeur de recopie.
6. Définir une fonction qui prend en argument un champ R et une matrice A et qui renvoie le champ dont chaque vecteur est le produit du vecteur de même indice du champ R et de la matrice A.
7. Tester toutes ces fonctions dans un programme principal.