Programmation Orienté Objet C++
Révisions

La Programmation Orientée Objet en C++ ajoute au langage C des concepts nouveaux et des notations supplémentaires.

Pour pouvoir se concentrer efficacement sur ces nouveautés, il sera indispensable d'avoir une bonne maitrise des bases du langage C (déclaration de variables, structures de contrôles *if, for, while*, etc.) et plus particulièrement des *struct* et des fonctions.

L'objectif de ce TD est de vous permettre de revoir ces notions, avant d'en apporter de nouvelles.

1 Divisible

On souhaite créer et manipuler des nombres que l'on va appeler **Divisible**.

Un nombre **Divisible** est un nombre dont on connait certains diviseurs. Dans le cas qui nous intéresse, on veut garder en mémoire le plus petit et le plus grand nombre premier qui divisent le Divisible.

Ce type d'information est utile dans certains protocoles cryptographiques dont la sécurité repose sur la factorisation de grands nombres entiers.

Prenons par exemple l'entier $5775 = 3 * 5 * 5 * 7 * 11$. Si on manipule le **Divisible** 5775, on retiendra que ses plus petit et plus grand diviseurs sont 3 et 11. Les valeurs 5 et 7 par contre n'ont pas à être enregistrées.

On remarquera que certaines valeurs ne peuvent pas être représentées par un **Divisible**, comme par exemple 0 ou 13. Dans les 2 cas, on serait amené à retenir comme plus petit diviseur une valeur qui n'est pas un nombre premier ($0 = 0 * 0$ et $13 = 1 * 13$, or 0 et 1 ne sont pas des nombres premiers)

1.1 Déclaration de la struct Divisible

Déclarer la struct **Divisible** possédant les attributs :

— **val** : en entier qui contient la valeur du Divisible (ex : 5775)

— **plus_petit_diviseur** : en entier qui contient le plus petit nombre premier qui divise val (ex : 3)

— **plus_grand_diviseur** : en entier qui contient le plus grand nombre premier qui divise val (ex : 11)

Ecrire un main() qui déclare une variable de type Divisible, donne des valeurs à ses trois attributs et les affiches.

1.2 Fonction afficher()

Créer une fonction qui affiche les attributs d'un **Divisible** passé en paramètre de la fonction.

Le main() suivant doit fonctionner :

```
int main()
{
Divisible A;
A.val = 5775;
A.plus_petit_diviseur = 3;
A.plus_grand_diviseur = 11;

afficher(A); // affiche 5775 (3..11)
}
```

1.3 Fonction creer()

Créer une fonction qui prend en paramètre deux nombres entiers (qu'on supposera premiers) et qui renvoie un **Divisible** valant le produit de ces 2 nombres.

Le main() suivant doit fonctionner :

```
int main()
{
Divisible A;
A = creer(3,17);
```

```
afficher(A); // affiche 51 (3..17)
}
```

1.4 Fonction multiplier()

Créer une fonction qui prend en paramètre deux **Divisible** et qui renvoie leur produit sous la forme d'un Divisible

```
int main()
{
Divisible A, B, C;

A=creer(7,13);
afficher(A); // affiche 91 (7..13)
B=creer(3,5);
afficher(B); // afiche 15 (3..5)

C=multiplier(A,B); // affiche 1365 (3..13)
afficher(C);
}
```

1.5 Autre fonction multiplier()

Créer une fonction qui prend en paramètre un entier (on supposera qu'il s'agit d'un nombre premier) et un **Divisible** et qui renvoie leur produit sous la forme d'un Divisible

```
int main()
{
Divisible A, B;

A=creer(7,13);
afficher(A); // affiche 91 (7..13)

B=multiplier(5,A);
afficher(B); // affiche 455 (5..13)
}
```

Remarque : comme il existe déjà une fonction nommée multiplier(), il est nécessaire d'utiliser un compilateur C++ et non un compilateur C pour pouvoir définir une autre fonction ayant le même nom mais des paramètres différents (on parle de surcharge de fonction).

1.6 Et encore une autre fonction multiplier()

Créer une fonction qui prend en paramètre un **Divisible** et un entier (on supposera qu'il s'agit d'un nombre premier), et qui renvoie leur produit sous la forme d'un Divisible.

Pour cette fonction, faire intelligement appel à la fonction précédente.

```
int main()
{
Divisible A, B;

A=creer(7,13);
afficher(A); // affiche 91 (7..13)

B=multiplier(A,5); // ATTENTION AUX PARAMETRES
afficher(B); // affiche 455 (5..13)
}
```

1.7 Bibliothèque divisible.h

Créer une bibliothèque regroupant les définitions de struct et de fonctions que vous avez écrit pour manipuler des Divisible. Prenez soin de bien séparer votre travail entre un fichier d'entête (divisible.h), un fichier contenant les corps des fonctions (divisible.cpp) et un fichier contenant un main() permettant de tester le bon fonctionnement de votre bibliothèque.

Si vous êtes sous Linux, et si vous savez le faire, créez également un fichier makefile.

2 Apports du C++

Puisque vous utilisez un compilateur C++, il est désormais possible d'utiliser certains ajouts du langage C++ par rapport au langage C.

2.1 Affichage et saisie

En C++, on utilise habituellement une bibliothèque d'entrée/sortie écrite en C++, à la place de la bibliothèque stdio.h

Voici un exemple :

```
#include <iostream>
using namespace std;

int main()
{
int n;

cout << "Quelle␣est␣la␣valeur␣?" << endl;
cin >> n;

cout << "le␣carre␣de␣<␣" << n << "␣vaut␣" << n*n ;
}
```

Inspirez vous de cet exemple pour que votre fonction $affiche r()$ utilise cette bibliothèque au lieu de la fonction printf().

Modifiez également votre main() pour saisir au clavier les valeurs associées à un Divisible que vous utilisez

2.2 Vector au lieu des tableaux

Le C++ permet également de manipuler des tableaux de maniée plus agréable qu'en C : on peut connaitre leur taille, ajouter des éléments, etc.

```
#include <iostream>
#include <vector>
using namespace std;

// recopier uniquement les valeurs paires d'un tableau
// dans un autre tableau et l'afficher
int main()
{
// creation de deux vectors d'entiers
vector<int> T = {2, 7, 4, 5, 3, 6};
vector<int> P;

// parcout de T
for (int i=0; i<T.size(); ++i)
  if (T[i]%2==0) // si T[i] est pair
    P.push_back(T[i]); // ajout d'elements à P

// parcours des valeurs de P
for (int val : P)
  cout << val << endl;
}
```

Inspirez vous de cette exemple pour déclarer un tableau d'entiers nommé *Premiers*, contenant les nombres premiers 2, 3, 5, 7, 11

Puis déclarez un tableau de Divisible nommé *Tab*, et remplissez le avec toutes les combinaisons possibles de deux valeurs distinctes de *Premiers* (c'est à dire (2,3), (2,5), (2,7), (3,5), etc.)

Enfin, utilisez une boucle pour afficher toutes les Divisible contenus dans *Tab*

2.3 References

Le C++ introduit la notion de référence, qui permet de simplifier certaines tâches qui nécessitent habituellement des pointeurs en C alors qu'on ne s'interesse pas *relleent* à la gestion de la mémoire.

Par exemple, en C, pour qu'une fonction modifie un des ses paramètres, on est obligé de donner à la fonction l'adresse mémoire de ce paramèetre.

```
void doubler_faux(int n)
{
  n=n*2;
}

void doubler_avec_pointeur(int *ptr)
{
  (*ptr)=(*ptr)*2;
}

void doubler_avec_reference(int &n)
{
  n=n*2;
}

int main()
{
int val=7;
cout << "initial␣:" << val << endl;

doubler_faux( val );
cout << "apres␣faux␣:␣" << val << endl;

doubler_avec_pointeur( &val ); // attention au &
cout << "apres␣pointeur␣:␣" << val << endl;

doubler_avec_reference( val );
cout << "apres␣reference␣:␣" << val << endl;
}
```

Testez ce programme afin de vérifier quelles fonctions arrivent effectivement à multiplier leur paramètre par 2.

Inspirez vous de cette technique pour écrire une fonction initialiser() qui permet d'écrire le main() suivant :

```
int main()
{
Divisible A;
initialiser(A);
afficher(A); // affiche 6 (2..3)
}
```

La valeur 6 est choisie par défaut, car elle correspond au plus petit nombre premier ayant 2 diviseurs distoncts.