Université Pierre et Marie Curie **Master M1 "Mathématiques et Applications"**
Période 2 **Programmation en C/C++**

TD 1 : Étude d'un code en C, compilation sous Unix et notions de bases sur les pointeurs

Considérons le programme suivant.

```
double integr_rectangle(double (*func)(double), double a, double b, int n){
        double delta,double_n,s,x=a;
        int i;
        double_n=n;
        delta=(b-a)/double_n;
        s=func(a);
        for (i=1;i<n;i++) {x+=delta; s+=func(x);}
        return s*delta;
}
```

1 *Étudier cette fonction et exécuter le code "à la main" en essayant de comprendre toutes les instructions.*

L'essentiel du programme ayant été écrit, il faut maintenant procéder aux formalités (ou à ce qui sera bientôt une formalité pour vous) de compilation du langage C. Il existe deux solutions.

Méthode A. Cette méthode découpe le programme en fichiers de trois types : le `.c` principal qui contient seulement la fonction `main()`, les `.h` d'en-tête et les `.c` qui contiennent le code des fonctions auxiliaires.

1. On crée un fichier `fichier1.c` qui contient la fonction `integr_rectangle`. Mais avant ça, il faut créer un fichier d'en-tête (header) qui définira les bibliothèques utilisées dans ces programmes (les `include`), les types définis (les `typedef`), les conventions (les `define`) et *auquel on se référera* lorsque l'on voudra utiliser `integr_rectangle` dans un programme exécutable. Votre fichier d'en-tête `fichier1.h` doit contenir les lignes suivantes :

   ```
   #include <stdio.h>
   #include <math.h>

   double integr_rectangle(double (*func)(double), double a, double b, int n);
   ```

   IL peut également contenir les commentaires, dûment introduits, de description des fonctions.
2. On écrit alors la fonction `integr_rectangle` dans un fichier `fichier1.c` qui doit commencer par

   ```
   #include "fichier1.h"
   ```

   On compile alors avec la commande

   ```
   gcc -c fichier1.c
   ```

   La compilation implique la création automatique d'un fichier "objet", appelé `fichier1.o`
3. Comme ce fichier ne comporte pas de commande principale (appelée `main`), donc ne peut pas être exécuté comme tel, on va devoir créer un fichier auxiliaire, qui servira uniquement à choisir les valeurs des paramètres `a,b` et `(*func)` et à exécuter la fonction `integr_rectangle` pour ces valeurs des paramètres. Voici un exemple de fichier source `test-fichier1.c` qui permet la compilation d'un tel programme.

   ```
   #include <stdio.h>
   #include <math.h>
   #include "fichier1.h"

   double  f(double x)
   {
       return (cos(x));
   }

   int     main(void)
   {
       double result;
       result = integr_rectangle(f, 0., 1.,100);
       printf("%lf\n", result);
       return 0;
   }
   ```

4. On compile alors ce dernier programme en tapant

   ```
   gcc fichier1.o test-fichier1.c -o test-fichier1 -lm
   ```

   Que signifie cette commande ? Par cette commande, on compile `test-fichier1.c` et crée un exécutable du nom de `test-fichier1` en même temps. La présence de `fichier1.o` indique que l'on va relier `test-fichier1.c` à l'objet `fichier1.o` (de façon à pourvoir en utiliser les fonctions). On aurait pu, ici, mettre plusieurs fichiers à relier :

   ```
   gcc fichier1.o fichier1prime.o test-fichier1.c -o test-fichier1 -lm
   ```

   La présence de `-lm` à la fin indique que l'on utilise la bibliothèque (`l` pour `library`) `m` (`m` est une bibliothèque de fonctions mathématiques, à laquelle on est obligé de faire appel dès qu'on utilise des fonctions trigonométriques, racine carrée, etc...). De façon plus générale, l'utilisation d'une bibliothèque `bibliotheque` autre que la bibliothèque standard dans la création de l'exécutable se signale avec l'ajout de `-lbibliotheque` à la fin de la ligne de création de l'exécutable.

5. On exécute avec la commande

   ```
   ./test-fichier1
   ```

   Remarque : la présence de `.` avant `/` signifie simplement que l'exécutable que l'on veut exécuter se trouve dans le répertoire courant. Si ce n'était pas le cas, il faudrait remplacer `.` par le chemin menant à l'exécutable en question.

   Remarque : en C++, toutes ces commandes fonctionnent de la même manière, à ceci près que l'on remplace `gcc` par `g++`.

Méthode B. La deuxième solution, plus compacte mais inadaptée à des programmes de plus grande ampleur, consiste à écrire toutes les fonctions (`integr_rectangle`, `f` et `void`) dans un même fichier `fichier2.c`, à la compiler avec la commande

```
gcc fichier2.c -o fichier2 -lm
```

puis à executer avec la commande

```
./fichier2
```

2 *Réaliser toutes ces étapes, pour chacune des deux méthodes.*

3 *De la même façon, écrire un programme* `fichier3.c` *qui calcule numériquement* $\int_a^b x^2 \sin(x) dx$ *pour des valeurs* a *et* b *données dans un premier temps dans le programme, puis, dans un deuxième temps par l'utilisateur en utilisant la fonction* `scanf` *décrite à la fin de ce TD.*

4 *Quels seront les effets des exécutions des programmes suivants ? Il est très important de comprendre l'intérêt des pointeurs et l'utilisation de* `&` *et* `*` *pour passer des valeurs aux adresses.*

```
#include <stdio.h>
#include <math.h>

int main(void){
        double *a;//Definit a comme un pointeur vers un double
        double c; //Definit l'emplacement memoire pour un double, appele c
        c=5.0; //Attribue au double c la valeur 5.0
        a=&c; //a devient l'adresse de c
        printf("%lf\n", *a);
        return 0;
}
```

```c
#include <stdio.h>
#include <math.h>

int main(void){
        double *a;
        double *b;
        double c=1.0,d=2.0,f,g;
        double *e;
        double *h;
        a=&c;
        b=&d;
        f=*a+*b;
        e=&f;
        g=(*a+*b+*e)/2.0;
        h=&g;
        printf("%lf\n", *h*2.0);
        return 0;
}
```

5 *Écrire une fonction* `exchange` *qui, appliquée à deux* pointeurs *vers des entiers, a pour effet d'en permuter les* valeurs *pointées. Écrire un programme qui utilise cette fonction sur un exemple simple et affiche les résultats.*

Comment se servir des fonctions d'entrée-sortie `puts, scanf, printf` ?

Les fonctions `puts, scanf, printf` sont d'un usage délicat, et une mauvaise écriture d'un programme y faisant appel peut ne pas être décelée lors de la compilation, et donc donner des résultats aberrants sans que l'on comprenne pourquoi. C'est pourquoi nous rappelons ici les bases de leur fonctionnement.

- *Rappels succincts sur la fonction* `puts` *:*

```c
puts ("bla bla");
```

imprime la chaine de caractères

```
bla bla
```

- *Rappels succincts sur la fonction* `scanf` *:*

```c
scanf("chaine", adresse1, adresse2,adresse3,..)
```

est une commande par laquelle le programme exécuté *attend* que l'utilisateur entre des données aux claviers (le plus souvent des paramètres du programme décidés à la dernière minute) : le programme fait une lecture formatée du flux standard d'entrée (le clavier par défaut). La `chaine` comporte des indications de format et la suite `adresse1, adresse2, adresse3,..` contient les adresses où seront affectées, dans l'ordre, les données.
Les indications de format se font comme suit. **En pratique, nous utiliserons `%lf` pour les `double` et `%d` pour les entiers**. Attention aux erreurs d'étourderie sur les conventions ! En théorie, on peut affiner cela avec la syntaxe `%[largeur][modificateur]type`, les arguments entre `[]` étant optionnels (plus de précisions sur cette syntaxes peuvent être trouvées sur internet).
Exemple :

```c
int i,j;
double x,y;
puts("Entrez les entiers i,j separes par une virgule");
scanf("%d, %d", &i, &j);
puts("Entrez maintenant les doubles x,y separes par une virgule");
scanf("%lf, %lf", &x, &y);
puts("Vous changez d'avis pour i et x ?
         Re-entrez i et x separes par un espace");
scanf("%d %lf", &i, &x);
```

- *Rappels succincts sur la fonction* `printf` *:*

```c
printf("chaine", arg1,arg2,...)
```

est une commande qui permet l'écriture formatée de `chaine` sur le flux standard de sortie stdout (l'écran par défaut). La chaîne de caractères peut contenir à la fois des caractères à afficher et des spécifications de format. Les instructions `\n` et `\t` y signifient respectivement "passer à la ligne" et "laisser un espace large (une tabulation)". Il devra y avoir autant d'arguments à la fonction `printf` qu'il y a de spécifications de format et elles seront lues dans l'ordre dans lesquelles elles apparaissent.
Les indications de format se font comme suit. **En pratique, nous utiliserons `%lf` pour les `double` et `%d` pour les entiers**. En théorie, on peut affiner cela avec la syntaxe `%[drapeaux][largeur][.precision][modificateur]type`, les arguments entre `[]` étant optionnels (plus de précisions sur cette syntaxes peuvent être trouvées sur internet). Exemple :

```
#include <stdio.h>
#include <math.h>

int main(void)
{
int i,j;
double x,y,id,jd;
puts("Entrez les entiers i,j separes par une virgule");
scanf("%d, %d", &i, &j);
puts("Entrez maintenant les doubles x,y separes par une virgule");
scanf("%lf, %lf", &x, &y);
id=(double) i;
jd=(double) j;
x+=id;
y+=jd;
printf("x+i vaut %lf\n",x);
printf("y+j vaut %lf\n",y);
return 0;
}
```