RAN Informatique – FA13

NOM : *Corrigé*	Prénom :
Filère :	

Seuls les polycopiés distribués en cours sont autorisés. Si vous manquez d'espace pour écrire utilisez la feuille d'examen fournie.
On suppose que ces programmes s'exécutent sur une p*lateforme 32 bits.*

Exercice 1

La fonction *strlen* , déclarée dans *string.h*, retourne la longueur d'une chaîne de caractères. Dans le décompte qu'elle effectue tous les caractères sont comptés, y compris les espaces. Sa signature est

```
int strlen(char *) ;
```

Écrire la fonction *strcharlen* qui effectue le même travail mais qui ne compte que les caractères alphabétique (de *a* à *z*, minuscule ou majuscule)

Exemple :

```
int main() {
  char s[100]="Mon Beau Sapin!";
  printf("strlen(\"%s\")=%d\n",s,strlen(s));
  printf("strcharlen(\"%s\")=%d\n",s,strcharlen(s));
  return 0;
}
```

donne le résultat suivant à l'écran (dans le cas où on a ajouté la définition demandée de *strcharlen*) :

```
strlen("Mon Beau Sapin!")=15
strcharlen("Mon Beau Sapin!")=12
```

```
// version où la chaîne s est vue comme un tableau de char
int strcharlen(char *s) {
  int n=0;
  for (int i=0;i<strlen(s);i++)
    if ((s[i]>='A' && s[i]<='Z') || (s[i]>='a' && s[i]<='z')) n++;
  return n;
}
// version où la chaîne s est vue comme un pointeur
int strcharlen(char *s) {
  int n=0;
  while (*s) {
    if ((*s>='A' && *s<='Z') || (*s>='a' && *s<='z')) n++;
    s++;
  }
  return n;
}
```

Exercice 2

La fonction ***n !*** est définie ainsi :$\forall n > 0$ $n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$ et ***0 ! =1***

- Implémenter une fonction *fact* en C conforme à ce pseudo code

```
int fact(int n) {
  int r=1;
  if (n==0) return r;
  else
    for(int i=1;i<=n;i++) r=r*i;  // on aurait pu mettre r*=i
  return r;
}
```

On souhaite tester cette fonction *fact* par le programme de test suivant :

```
int fact(int n) {
  ...
}
int main() {
 int n;
 scanf("%d",&n);
 printf("fact(%d)=%d\n",n,fact(n));
 return 0;
}
```

L'utilisateur teste le programme suivant en entrant successivement les valeurs suivants : 5, 10 et 20
Les résultats sont les suivant

```
colin@colin-desktop:~/module/RN_INFO/2010$ ./exoqroc1
5
fact(5)=120
colin@colin-desktop:~/module/RN_INFO/2010$ ./exoqroc1
10
fact(10)=3628800
colin@colin-desktop:~/module/RN_INFO/2010$ ./exoqroc1
20
fact(20)=-2102132736
```

- Pouvez-vous expliquer l'anomalie de ce dernier résultat ?

factorielle(20) donne un résultat aberrant car on a dépassé la capacité maximale de représentation du type *int*.
Sur une architecture 32 bits la taille d'un *int* est également 32 bits. Un entier peut donc représenter 2^{32} valeurs différentes soit un intervalle de valeurs entières signées de -2 milliard à +2 milliard environ.
20 ! a une valeur supérieure à cette limite.

- Après avoir analysé le problème précédent vous décidez de ne pas modifier votre code mais de munir votre fonction d'une documentation précisant les limites d'utilisation de la fonction. Déduire des valeurs obtenues ci-dessus pour n valant *5*, puis *10* et *20* la limite exacte maximale concernant ce

paramètre au delà de laquelle le calcul effectué par *fact* donnera un résultat erroné.

D'après les résultats ci-dessus 10!=3 628 800
11!=10 ! * 11 ≈ 40 000 000
12!=11 ! * 12 ≈ 480 000 000
13!=12 ! * 13 ≈ 480 000 000 * 13 ≈ 5 000 000 000
Cette dernière valeur excède la limite de + 2 milliard. On voit donc que cette limite est dépassée pour n = 13.
Dans le mode d'emploi de la fonction fact fournie il faudra préciser que pour être utilisée valablement le paramètre n devra se situer dans l'intervalle [0-12]

Exercice 3

Les complexes sont modélisés ci-dessous comme des entités comprenant 2 champs réels appelés *partie réelle (re)* et *partie imaginaire (im)*.

Le programme suivant crée 3 nombres complexes, et les affiche à l'écran sous une forme qui utilise l'imaginaire *i*

```
#include <stdio.h>
struct Complexe {
  double re,im;
};
void print(Complexe c) {
  if (c.im==0) printf ("%lg",c.re);
  else if (c.im<0) printf("%lg - i*%lg",c.re,-c.im);
  else printf("%lg + i*%lg",c.re,c.im);
}
int main() {
  Complexe c1,c2,c3;
  c1.re=5.6;c1.im=-2.3;
  c2.re=8.7;c2.im=0;
  c3.re=3.4;c3.im=7.3;
  print(c1);printf("\n");
  print(c2);printf("\n");
  print(c3);printf("\n");
}
```

- Sachant que *printf(«%lg», 2.3)* affiche *2.3* et que *printf(«%lg», - 2.3)* affiche *-2.3*, indiquer quel sera exactement l'affichage produit par ce programme.

5.6 – i*2.3
8.7
3.4 + i*7.3

Le conjugué du nombre complexe $z = a + ib$est $\bar{z} = a - ib$. On se propose de créer et de tester dans le progamme ci-dessous la procédure *conjugue* qui transforme un complexe en son conjugué. On ajoute donc la procédure conjugue au programme précédent et le code de test ci-dessous

```
...
void conjugue(Complexe c) {
    c.im=-c.im;
}

int main() {
  Complexe c1,c2,c3;
  c3.re=3.4;c3.im=7.3;
  print(c3);
  printf("\n");
  conjugue(c3);
  print(c3);
  printf("\n");
}
```

• Le test est un échec : l'affichage de *c3* est identique avant et après l'application de la procédure *conjugue*. Pouvez-vous expliquer ce résultat ?

Dans l'appel de la procédure *conjugue(c3)*, le paramètre *c3* est passé par valeur. Cela signifie qu'il sera recopié dans la pile pour être traité par la procédure *conjugue*. Cette dernière va donc travailler sur une copie de *c3* est non sur la variable *c3* elle même, et c'est donc cette copie qui sera modifiée (puis détruite juste après...!) et non *c3*.
Dans l'espace d'appel la variable c3 reste donc inchangée.

• Transformer la procédure *conjugue* et le test de mise en œuvre pour que le résultat soit conforme à ce qu'on attend.

```
void conjugue(Complexe * pc) {
    (*pc).im=-(*pc).im; // on aurait pu écrire pc->im=-pc->im
}

// au niveau de l'appel le code ci-dessus devient
int main() {
  Complexe c1,c2,c3;
  c3.re=3.4;c3.im=7.3;
  print(c3);
  printf("\n");
  conjugue(&c3);
  print(c3);
  printf("\n");
}
// certains ont fait une fonction à la place de la procédure. C'est également correct.
Complexe conjugue(Complexe c) {
    c.im=-c.im;
    return c;
}
// au niveau de l'appel le code ci-dessus devient cette fois
int main() {
  Complexe c1,c2,c3;
```

```
        c3.re=3.4;c3.im=7.3;
        print(c3);
        printf("\n");
        c3=conjugue(c3);
        print(c3);
        printf("\n");
      }
```

Cette structure *Complexe* est adaptée aux problèmes où la représentation *partie réelle/partie imaginaire* convient. Dans certains cas cependant il serait plus maniable d'avoir des nombres complexes représentés par le couple (ρ,θ)désignant respectivement le module du complexe (dans $\mathbb{R}^+$) et son angle (dans *[0..2π[*)
Dans ce but on crée le type *PolComplexe* défini ainsi :

```
struct PolComplexe {
  double rho,theta;
};
```

qu'on complète par la procédure d'affichage *printPol* suivante :

```
void printPol(PolComplexe cp) {
  printf("(%lg,%lg)",cp.rho,cp.theta);
}
```

On considère le programme de test suivant et son résultat à l'écran.

```
int main() {
  Complexe c;
  PolComplexe cp;
  c.re=1;c.im=1;
  print(c);printf("\n");
  cp=car2Pol(c);
  printPol(cp);printf("\n");
}
```

L'affichage est le suivant :

```
1 + i*1
(1.41421,0.785398)
```

- Que représente les nombres *1.41421* et *0.785398* ?

Respectivement le module du complexe (1,1), c'est à dire $\sqrt{2}$, et l'angle correspondant c'est à dire $\frac{\pi}{4}$

- Écrire complètement une fonction *car2Pol* qui est compatible avec l'usage présenté ci-dessus

```
PolComplexe car2Pol(Complexe c) {
  PolComplexe pc;
  pc.rho=sqrt(c.re*c.re+c.im*c.im);
  pc.theta=atan(c.im/c.re);
  return pc;
}
```

- Ecrire la procédure *conjuguePol* qui transforme un *PolComplexe* en son conjugué. Attention il

faut maintenir la contrainte $\theta \in [0,2\pi[$. La constante π peut être manipulée en *C* au moyen de la macrodéfinition *M_PI*

```
// version procédurale
void conjuguePol(PolComplexe *ppc) {
    ppc->theta=M_PI+ppc->theta;
    if (ppc->theta>=2*M_PI) ppc->theta-=2*M_PI;
}
// appel
  PolComplexe cp;
  c.re=1;c.im=1;
  print(c);printf("\n");
  cp=car2Pol(c);
  printPol(cp);printf("\n");
  conjuguePol(&cp);
  printPol(cp);printf("\n");
-------------------------------------
// version fonctionnelle
PolComplexe conjuguePol(PolComplexe pc) {
    pc.theta=M_PI+pc.theta;
    if (pc.theta>=2*M_PI) pc.theta-=2*M_PI;
    return pc ;
}
// appel
  PolComplexe cp;
  c.re=1;c.im=1;
  print(c);printf("\n");
  cp=car2Pol(c);
  printPol(cp);printf("\n");
  cp=conjuguePol(cp);
  printPol(cp);printf("\n");
```