1°/ Les bonnes pratiques : Une Solution

1.1.1/ Tester le premier programme C

Il faut savoir que

- ✓ **argc est un entier qui désigne le nombre d'arguments de la fonction main**
- ✓ **argv[i] est une chaine de caractères qui désigne l'argument de rang i (i ≥ 0)**

1. **Copier/coller le texte sous-dessous sous le nom de "exo0.c"**
2. **Changer 1 par 0 dans : for(i=1… par for(i=0…**
3. **Ajouter l'affichage de argc**

```
#include <stdio.h>
int main(int argc, char** argv) {
int i;
printf("Le nombre d'arguments est : %d\n",argc) ;
for(i=0;i<argc;i++) print("%s\n",argv[i]);
return 0;
}
```

4. **Compilation : gcc –c –Wall exo0.c ➔ exo0.o**
5. **Corriger l'erreur "print" en "printf" sur la ligne n°5**
6. **Recompilation : gcc –c –Wall exo0.c ➔ exo0.o (Création du fichier objet)**
7. **Edition de liens : gcc exo0.o -o exo0 ➔ exo0 (Création du fichier exécutable)**
8. **Exemple d'exécution :**
   ./exo0 Ceci est un test

   Le nombre d'arguments est : 5

   ./exo0 ➔ est le contenu de argv[0]
   Ceci ➔ est le contenu de argv[1]
   est ➔ est le contenu de argv[2]
   un ➔ est le contenu de argv[3]
   test ➔ est le contenu de argv[4]

1.1.2/ Tester les paramètres

1. **Copier/coller le texte sous-dessous sous le nom de "exo1.c"**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void usage() {
  fprintf(stderr,"usage : myprog ip port\n\t- ip : A.B.C.D (with 0<A,B,C,D<255)\n\t- port : from
0 to 65535\n");
  exit(1); // termine l'exécution
}

int main(int argc, char**argv) {
  if (argc != 3) usage();
  if ((strlen(argv[1]) < 7) || (strlen(argv[1]) > 15)) usage();
  int port = atoi(argv[2]);
  if ((port < 0) || (port > 65535)) usage();
  printf("ip : %s, port : %d\n",argv[1],port);
  return 0;
}
```

Ce programme utilise les fonctions suivantes : fprintf, exit, strlen et atoi

- ✓ **Pour fprintf et exit voir votre cours**
- ✓ **Pour les autres :**
  Il faut utiliser la commande : whereis nom_de_la_fonction pour avoir le n° 3 du manuel
  Il faut utiliser la commande : man 3 nom_de_la_fonction
  À vous de lire les 3 paragraphes :
  Nom de la fonction
  Synopsis pour avoir le fichier d'entête à inclure et le prototype
  Description pour avoir l'effet de la fonction

Pour strlen:

NOM strlen - Calculer la longueur d'une chaine de caractères
SYNOPSIS #include <string.h>
size_t strlen(const char *);
DESCRIPTION La fonction strlen() calcule la longueur de la chaine de caractères sans compter l'octet nul (" \0") final.

Pour atoi:

NOM atoi, atol, atoll, atoq - Convertir une chaine en entier
SYNOPSIS #include <stdlib.h>
DESCRIPTION La fonction atoi() convertit le début de la chaine pointée par nptr en entier de type int.

1. **Compilation : gcc –c –Wall exo0.c ➔ exo0.o (Création du fichier objet)**
2. **Edition de liens : gcc exo0.o -o exo0 ➔ exo0 (Création du fichier exécutable)**
3. **A vous de tester les exemples ci-dessous.**
   exo1
   exo1 1.2.3.4
   exo1 1.2.3.4 -10
   exo1 1.2.3.4 21
   exo1 1.2.3.4.5.6 21

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void usage() {
  fprintf(stderr,"usage : myprog ip port\n\t- ip : A.B.C.D (with 0<A,B,C,D<255)\n\t- port : from
0 to 65535\n");
  exit(1); // termine l'exécution
}

int main(int argc, char**argv) {
  if (argc != 3) usage();
  if ((strlen(argv[1]) < 7) || (strlen(argv[1]) > 15)) usage();
  int port = atoi(argv[2]);
  if ((port < 0) || (port > 65535)) usage();

 int p1,p2,p3,p4;
  int nbOk = sscanf(argv[1],"%d.%d.%d.%d",&p1,&p2,&p3,&p4);
  if (nbOk != 4) usage();
  if ((p1<0)||(p1>255)||(p2<0)||(p2>255)||(p3<0)||(p3>255)||(p4<0)||(p4>255)) usage();

  printf("ip : %d.%d.%d.%d, port : %d\n",p1,p2,p3,p4,port);
  return 0;
}
```

NOM **scanf** : Entrées formatées
SYNOPSIS : #include <stdio.h
int scanf(const char *format, ...);
DESCRIPTION : Les fonctions de la famille scanf() analysent leurs entrées conformément au format décrit plus bas. Ce des indicateurs de conversion. Les résultats des conversions, s'il y en a, sont stockés dans des endroits pointés par des arguments pointeurs qui suivent le format. Chaque argument pointeur doit être du type approprié pour la valeur retournée par la spécification de conversion correspondante.
Si le nombre de spécifications de conversion dans format excède le nombre d'arguments pointeur, le résultat est indéterminé. Si le nombre d'arguments pointeur excède le nombre de spécifications de conversion, les arguments pointeur en excès sont évalués mais ignorés. La fonction scanf() lit ses données depuis le flux d'entrée standard stdin, fscanf() lit ses entrées depuis le flux pointé par stream, et sscanf() lit ses ent ées dans la chaîne de caractères pointée par str.

Remarque :

Même avec la solution ci-dessus, le dernier exemple ne marche pas :

exo1 1.2.3.4.5.6 21
Pour corriger cette dernière, plusieurs solutions existent.
- On peut compter le nombre de points. S'il y 4 on accepte.
- La plus simple consiste à utiliser sscanf avec lecture formatée sur 5 variables, mais accepté uniquement le retour de la fonction avec l'entier 4
Changer ceci :

```
int p1, p2 , p3 , p4 ;

int nbOk = sscanf(argv[1],"%d.%d.%d.%d",&p1,&p2,&p3,&p4);

if (nbOk != 4) usage();
```

par :

```
int p1 , p2 , p3 , p4 , p5 ;

int nbOk = sscanf(argv[1],"%d.%d.%d.%d.%d",&p1,&p2,&p3,&p4,&p5);

if (nbOk != 4) usage();
```

1.2°/ Compiler avec l'option –Wall

Modifiez le programme comme suivant :

```
#include <stdio.h>
void afficheDate(int day, int month, int year) {
  if (year == 0) return;                    //Correction : ==
  printf("Nous somme le %d/%d/%d\n",day,month,year);
return ;
}

int main(int argc, char** argv) {
  int day,month,year;
  afficheDate(day,month,year);
}
```

```
int main(int argc, char** argv) {
  int i=0;
  int day,month,year;
day = month = year= 0 ;            //Correction : Initialisation
  afficheDate(day,month,year);
return 0 ;                         //Correction : Le fonction main retourne 0
}
```

- Recréez l'exécutable et lancez-le. Cette fois, on obtient un résultat différent. Par exemple : Nous sommes le 0/32767/0
- On remarque donc que le compilateur ne donne pas de valeur par défaut aux variables, puisqu'elles ne sont toujours pas initialisées et pourtant la valeur d'une d'entre-elles à changer par rapport au premier test.

Question 1 : comment est-il possible que l'ajout d'un simple variable (i en l'occurrence) dans la source puisse modifier la valeur que contient les variables non initialisées ?

Réponse : quand on déclare une variable, le compilateur ne va pas lui donner de valeur par défaut comme en Java. Lors de l'exécution, la valeur de cette variable (locale à la fonction main) sera stockée dans un emplacement mémoire appelé la pile. Or, lorsque l'on commence l'exécution d'un programme, sa pile n'est pas initialisée non plus et elle utilise la mémoire physique en l'état. Il peut donc y avoir déjà des valeurs différentes de 0 en mémoire (même si la plupart du temps il y a effectivement des 0). Si on déclare une variable de plus (en l'occurrence i), le compilateur va prévoir de l'espace supplémentaire sur la pile pour contenir sa valeur. Généralement, le compilateur prévoit l'espace dans l'ordre où sont déclarées les variables. Comme on ajoute i avant les autres, elles sont "décalées" en mémoire, ce qui explique qu'elles puissent avoir tout d'un coup une valeur différente de 0.
À titre d'essai, vous pouvez ajouter encore une déclaration juste après i (par exemple int j=0) et vous constaterez que cette fois day et month ont une valeur.

1.4°/ Mettre les bons types de variable

1.4.1°/ opérations entre deux types différents.

- Pour constater l'effet de telles opérations, copiez/collez le code suivant dans un fichier nommé exo2.c. Il permet de crypter une phrase en décalant les lettres d'une certaine valeur (par ex. a +2 donne c, b+2 donne d, ...). (NB : c'est le cryptage dit "de César", qu'il aurait employé lors de ses campagnes militaires).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int main(int argc, char** argv) {
  if (argc != 3) exit(1);
  int i;
  int decal = atoi(argv[1]); // transforme la chaîne de caractères dans argv[1] en un int
  char* msg = argv[2]; // crée une variable qui est un alias de la chaîne de caractères dans argv[2]
  for(i=0;i<strlen(msg);i++) {
```

```
    msg[i] += decal; // ajoute decal au code ASCII de chaque élément de msg
  }
  printf("message crypté : %s\n",msg);
  return 0;
}
```

- Compilez le code : gcc -c -Wall exo3.c. On remarque qu'il n'y a aucun warning, malgré la présence de -Wall
- Faîtes l'édition de liens : gcc exo3.o -o exo3

- Le tableau ci-dessous donne dans la colonne de gauche la ligne pour exécuter exo3 avec différents paramètres. Reportez dans la colonne de droite le résultat.

commande	résultat
exo3 1 abcdef	bcdefg
exo3 10 abcdef	klmnop
exo3 128 abcdef	des caractères bizarres
exo3 256 abcdef	abcdef
exo3 257 abcdef	bcdefg
exo3 65536 abcdef	abcdef

- On constate qu'un décalage de 256 ou 65536 équivaut à 0, et 257 à 1.
- En effet, si on prend le code ascii de 'a', cela correspond à la valeur 97. Un char étant sur 1 octet, la valeur maximale non signée est 255. On peut donc ajouter au maximum 255-97 = 158 sans avoir de débordement. Mais si on ajoute 256, cela dépasse les 158 possibles. Comme on l'a vu plus haut, un débordement revient à "boucler". Donc si on ajoutait 159, on obtiendrait la valeur 0, si c'était 159, cela serait 1, ...Et si on continue ainsi jusqu'à 256, on obtient la valeur ... 97 !
- On en déduit également que tout multiple de 256 revient à boucler. Par exemple, si on relance exo3 avec un décalage de 512, 768, 1024, ..., 65536, 65792, ..., on boucle et on obtient tout le temps abcdef.

Question 2 : sans exécution, trouvez ce que va renvoyer exo3 65535 abcdef.

Réponse : d'après les remarques ci-dessus, on sait que 65536 est un décalage qui fait boucler, qui est donc similaire à ajouter 256. Dans la question 2, le décalage fait 65535 (c.a.d. 65536 - 1), donc c'est comme si le décalage était de 256 - 1 = 255. Si on prend le code ASCII de 'a', à savoir 97 et qu'on lui ajoute 255, on obtient 352, ce qui dépasse 256. Pour retrouver la valeur sans dépassement, il suffit de retrancher 256, donc on obtient 96. Conclusion : un décalage de 255 correspond en fait à un décalage de -1. Grâce à la table ascii, on en déduit que la réponse à la question 2 est `abcde

Question 3 : sans exécution, trouvez ce que va renvoyer exo3 2147483650 abcdef.

Réponse : comme pour la question 2, on peut facilement rapporter ce décalage à un se trouvant entre 0 et 256. Cela se fait simplement avec le reste de la division entre 2147483650 et

256 (c.a.d. en notation info 2147483650 % 256). On trouve facilement que cela donne 2. La réponse à la question 3 est donc : cdefgh

8 388 608

On pose ASCII(car) = n avec $0 \leq n \leq 255$
Soit $d \in N$ / Decalage(car,d) = ASCII(car) + d
On peut écrire : $\exists$ k,r / ASCII(car) + d = 256k +r avec $0 \leq r \leq 255$
Donc Decalage(car,d) = r ou
Decalage(car,d) = (ASCII(car) + d) % 256 (a% b : est le reste de la division de a par b)

1.4.2°/ lors de l'appel de fonctions

- Pour constater l'effet que peux avoir une variable du "mauvais type" lors de l'appel à une fonction, copiez/coller le code suivant dans un fichier nommé exo4.c

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

unsigned short division(unsigned short num, unsigned short denum) {
  return num/denum;
}
int main(int argc, char** argv) {
  if (argc != 3) exit(1);
  int n = atoi(argv[1]);
  int d = atoi(argv[2]);
  printf("%u\n",division(n,d));
  return 0;
}
```

- Compilez le code : gcc -c -Wall exo4.c. On remarque qu'il n'y a aucun warning, malgré la présence de -Wall
- Faîtes l'édition de liens : gcc exo4.o -o exo4

- Le tableau ci-dessous donne dans la colonne de gauche la ligne pour exécuter exo3 avec différents paramètres. Reportez dans la colonne de droite le résultat.

commande	résultat
exo4 10 100	0
exo4 100 100	1
exo4 1000 100	10
exo4 10000 100	100
exo4 100000 100	344
exo4 1000000 100	169

- D'après les résultats, on voit clairement qu'il y a un problème à partir de 100000.
- La raison en est très simple : le code fait une division entre deux valeurs unsigned short. Le premier paramètre du programme représente le numérateur qui est stockée temporairement dans un int nommé n. Mais lorsque l'on appelle la fonction, cet int est interprété comme un short, qui plus est non signé. Cela veut dire que la fonction ne tient compte que des deux derniers octets de l'int, les deux autres étant "jetés". Il va de soit que la fonction manipule ainsi une valeur potentiellement différente de l'originale.
- D'où une question : comment savoir si le résultat sera invalide ? Tout simplement en se référant aux valeurs limites pour les types.
- Par exemple, un unsigned short est limité à 65535. Toute valeur supérieure donnera un résultat incohérent.

Question 4 : sans exécution, trouvez ce que va renvoyer exo4 65536 10.

Réponse : quand on stocke 65536 dans un entier sur 32 bits, cela correspond en binaire (de gauche à droite) à quinze zéros, puis un 1, et enfin seize 0. Quand on passe cet int en paramètre à la fonction de division, seuls les 16 bits de poids faible (c.a.d ceux qui sont à droite) sont conservés. Or, ces 16 bits sont tous à zéro, ce qui correspond donc à la valeur numérique 0. La réponse à la question 4 est donc 0 divisé par 10, soit 0.

Question 5 : sans exécution, trouvez ce que va renvoyer exo4 65736 10.

Réponse : en binaire sur 32 bits, 65736 s'écrit (de gauche à droite) avec quinze 0, puis un 1, puis 0000000011001000. Si on ne garde que ces 16 derniers bits, cela correspond à la valeur 200. La réponse à la question 5 est donc 200/10 = 20.

2°/ Exercice applicatif

- Les mauvaises pratiques fondamentales :
  - pas de test des paramètres du programme,
  - pas d'initialisation des variables et notamment h2, m2, s2 avec 23:40:36
  - pas de test de la valeur des variables dans les fonctions.
  - dans gapInSecond(), si le deuxième horaire est antérieur, on obtient un écart négatif, ce qui ne va pas avec l'unsigned short. Il faut donc prendre la valeur absolue de l'écart.
  - de plus, l'écart maximal entre 2 horaires est de 23 heures, 59 minutes, 59 secondes, ce qui donne 86399 secondes. C'est une valeur qui dépasse le maximum d'un unsigned short, à savoir 65535. Il faut donc modifier la fonctionpour qu'elle renvoie un unsigned int.

  - addSeconds prend en paramètre un short comme nombre de secondes, ce qui est peut-être trop "petit". On s'en aperçoit par exemple quand on ajoute 46861 s et que l'on obtient un résultat aberrant. Un int conviendrait mieux.

  - pas de %24 lors du calcul de l'heure dans addSeconds.

- Pour les fautes de frappe :
  - le dernier paramètre de gapInSecond() devrait être char s2,
  - hor2sec utilise m1 au lieu de m2

- la soustraction se fait entre les deux mêmes valeurs ce qui produit toujours 0,
- dans addSeconds(), l'affectation des minute se fait dans hour[2] au lieu de hour[1].

- Après modification, on obtient par exemple :

```
#include <stdio.h>
#include <stdlib.h>
void usage() {
  fprintf(stderr,"usage : exo6 h m s nb_sec\n");
  exit(1);
}
/* gapInSecond() : renvoie l'écart en secondes entre deux horaires
   donnés sous la forme h1:m1:s1 et h2:m2:s2
 */
unsigned int gapInSecond(char h1, char m1, char s1, char h2, char m2, char s2) {
  /* NB : si les paramètres sont invalides, on choisit de les corriger
     à la valeur valide la plus proche.
  */
  if (h1<0) h1 = 0;
  else if (h1>23) h1 = 23;
  if (h2<0) h2 = 0;
  else if (h2>23) h2 = 23;
  if (m1<0) m1 = 0;
  else if (m1>59) m1 = 59;
  if (m2<0) m2 = 0;
  else if (m2>59) m2 = 59;
  if (s1<0) s1 = 0;
  else if (s1>59) s1 = 59;
  if (s2<0) s2 = 0;
  else if (s2>59) s2 = 59;

  unsigned int gap;
  int hor1Sec;
  int hor2Sec;
  hor1Sec = h1*3600 + m1*60 + s1;
  hor2Sec = h2*3600 + m2*60 + s2;
  gap = hor2Sec - hor1Sec;
  if (hor2Sec >= hor1Sec)   gap = hor2Sec - hor1Sec;
  else   gap = hor1Sec - hor2Sec;
  return gap;
}
```

```
/* addSeconds() : ajoute un nombre de secondes nbSec à un horaire donné
   sous la forme d'un tableau d'octets représentant h:m:s.
   Ce sont ces mêmes h, m et s qui vont être modifiés par la fonction
 */
void addSeconds(char* hour, int nbSec) {

  /* NB : si le contenu de hour est invalide, on choisit de le corriger
     à la valeur valide la plus proche.
  */
  if (hour[0]<0) hour[0] = 0;
  else if (hour[0]>23) hour[0] = 23;
  if (hour[1]<0) hour[1] = 0;
  else if (hour[1]>59) hour[1] = 59;
  if (hour[2]<0) hour[2] = 0;
  else if (hour[2]>59) hour[2] = 59;
  if (nbSec < 0) nbSec = 0;

  int horSec;
  horSec = (hour[0]) * 3600 + (hour[1]) * 60 + (hour[2]) + nbSec;
  hour[0] = (horSec/3600)%24;
  hour[1] = (horSec%3600)/60;
  hour[2] = (horSec%3600)%60;
}

int main(int argc, char **argv) {
  int h1=0,h2=0,m1=0,m2=0,s1=0,s2=0;
  int nbS = 0;
  char hour[3];
  if (argc != 5) usage();
  h1 = atoi(argv[1]);
  m1 = atoi(argv[2]);
  s1 = atoi(argv[3]);
  nbS = atoi(argv[4]);

  if ((h1<0)||(m1<0)||(s1<0)||(h1>23)||(m1>59)||(s1>59)) usage();
  if (nbS < 0) usage();
  hour[0] = h1; hour[1] = m1; hour[2] = s1;
  h2 = 23; m2 = 40; s2 = 36;
  // ne pas modifier les 5 lignes suivantes
  int g = gapInSecond(h1,m1,s1,h2,m2,s2);
```

```
  printf("écart entre %d:%d:%d et 23:40:36 = %d\n",h1,m1,s1,g);
  printf("%d:%d:%d augmenté de %d secondes donne",hour[0],hour[1],hour[2],nbS);
  addSeconds(hour,nbS);
  printf(" %d:%d:%d\n",hour[0],hour[1],hour[2]);
  return 0;
}
```