Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 01 | entrées-sorties

Présentation

Le but de ce TP est d'apprendre :

- Comment écrire un programme simple, comportant seulement une fonction principale ;
- Comment afficher du texte à l'écran et saisir du texte entré par l'utilisateur.

1 Organisation du code source

Les programmes en langage C sont structurés au moyen de *fonctions*. Cependant, celles-ci seront étudiées plus tard, et on ne les utilisera donc pas dans les premiers TP. Pour cette raison, le code source de ces premiers TP aura une organisation particulière, de la forme suivante :

```
int main()
{  // exercice 1
   ...   // code source de l'exercice 1 ...

   // exercice 2
   ...   // code source de l'exercice 2 ...

   ...   // reste des exercices

   return EXIT_SUCCESS;
}
```

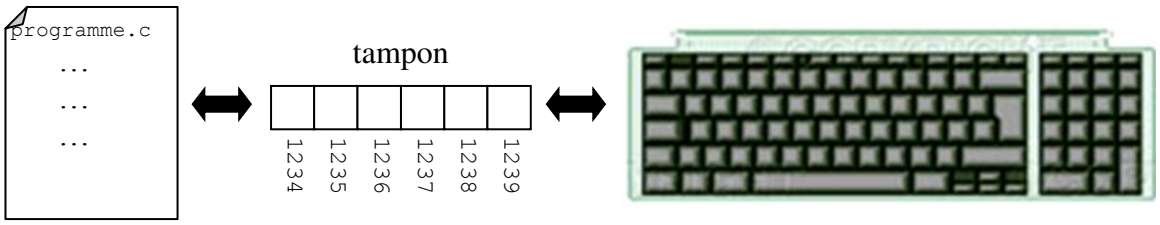
Autrement dit, la fonction `main` contiendra tous les exercices, identifiés par des commentaires appropriés. Quand un exercice sera terminé, le code source correspondant sera lui-même placé en commentaire, afin de ne pas interférer avec l'exercice suivant.

2 Notion d'entrées-sorties

Le terme d'entrées/sorties désigne les mécanismes utilisés pour qu'un programme puisse recevoir/envoyer des données depuis/vers un **flux**. Un flux peut être un fichier, un périphérique (écran, clavier, etc), ou d'autres entités. Dans ce TP, nous nous intéressons seulement à l'entrée **standard**, qui est le clavier, et à la sortie **standard**, qui est l'écran. En langage C, le flux d'entrée standard est appelée `stdin` (pour *st*andar*d in*put) et le flux de sortie standard est appelée `stdout` (pour *st*andar*d out*put).

Il existe de nombreuses méthodes permettant d'effectuer des entrées/sorties en langage C, suivant le type des données à traiter. On distinguera les fonctions qui manipulent les données de manière **non-formatée**, c'est-à-dire caractère par caractère, et celles qui travaillent de manière **formatée**.

Par défaut, l'accès au flux est **bufférisé**, ce qui signifie qu'une zone de mémoire sert de tampon. Par exemple, dans le cas du flux d'entrée standard, les données issues du clavier sont **stockées** dans le tampon, et **attendent** que le programme ait besoin d'elles.



L'intérêt de l'accès bufférisé est que, tant que le programme ne demande pas l'accès au tampon, il est possible de **modifier** les données qui y sont contenues.

Remarque : pour pouvoir utiliser les fonctions d'entrée-sortie présentées dans ce TP, vous devez inclure la bibliothèque `stdio.h` dans votre programme.

3 Entrées/sorties non-formatées

La fonction `int getc(FILE *stream)` renvoie le code ASCII du caractère qui a été lu dans le flux `stream`. Si on veut saisir un caractère au clavier, on écrira :

```
char c;
c = getc(stdin);
```

Si une erreur se produit, la fonction renvoie `-1`. On ne tiendra pas compte de cette possibilité lorsqu'on travaille sur le flux standard `stdin`, mais il est important de connaître cette propriété, qui sera utile plus tard.

La fonction `getchar` fonctionne exactement pareil que `getc`, à la différence qu'on n'a pas à préciser qu'on utilise le flux standard :

```
char c;
c = getchar();
```

Ces deux fonctions `getc` **prennent** un caractère dans le tampon et le **suppriment** du tampon. Si le tampon est **vide**, le programme se **bloque** jusqu'à ce qu'un `char` soit placé dans le tampon, c'est-à-dire : jusqu'à ce qu'une valeur soit saisie au clavier.

La fonction `int putc(int c, FILE *stream)` écrit le caractère dont le code ASCII est `c` dans le flux `stream`. Si on veut afficher un caractère à l'écran, par exemple le caractère `A`, on écrira donc :

```
char c = 'A';
putc(c, stdout);
```

Si une erreur se produit, la fonction renvoie `-1`, sinon elle renvoie le code ASCII du caractère qui a été écrit. Là encore, on ne tiendra pas compte, ici, de la possibilité d'erreur. La fonction `putchar` fonctionne exactement pareil que `putc` sans préciser le flux :

```
char c = 'A';
putchar(c);
```

Exercice 1

En utilisant `getchar` et `putchar`, écrivez un programme qui saisit un caractère et qui l'affiche à l'écran.

Exercice 2

Même question avec deux caractères : d'abord le programme saisit les deux caractères, puis il affiche les deux caractères.

Remarque : pour changer de ligne, vous pouvez afficher le caractère `'\n'`.

4 Sorties formatées

La fonction `putchar` ne permet d'afficher que des caractères. La fonction `printf`, elle, permet d'effectuer une sortie formatée, et d'afficher des valeurs de n'importe quel type simple :

```
printf(format,exp1,...,expn);
```

La fonction affiche les valeurs des expressions `exp1,...,expn`. Le paramètre `format` est une **chaîne de caractère de formatage** indiquant **comment** les expressions doivent être affichées. Cette chaine de caractères peut contenir deux sortes d'informations :

- du texte **normal**, qui sera affiché tel quel
- du texte spécifiant des **formats** d'affichage

Par exemple, pour afficher le texte normal *bienvenue dans le programme*, on fera (dans l'exemple, la ligne foncée représente le résultat à l'écran) :

```
printf("bienvenue dans le programme");
bienvenue dans le programme
```

Un format d'affichage prend la forme d'un **pourcent** `%` suivi d'une expression précisant le format. Par exemple, le format `%d` signifie que l'on veut afficher un entier relatif exprimé en base 10 (entier décimal) :

```
printf("%d",1234);
1234
```

Il est possible d'avoir du texte normal et des formats dans la **même** chaîne de formatage :

```
printf("affichage du nombre : %d",1234);
affichage du nombre : 1234
```

Lors de l'affichage, chaque format présent dans la chaine de formatage est utilisé pour afficher la valeur d'une des expressions passées en paramètre. Le 1er format sert à afficher la 1ère expression, le 2ème format à afficher la 2ème expression, etc. :

```
printf("affichage du nombre 1 : %d. affichage du nombre 2 : %d",12,34);
affichage du nombre 1 : 13. affichage du nombre 2 : 34
```

On manipule des **expressions**, on n'est donc pas limité à des **constantes littérales**, on peut utiliser des **variables** et des **opérateurs**, par exemple :

```
int a=5;
printf("%d plus %d egale %d",10,a,10+a);
10 plus 5 egale 15
```

Remarque : le caractère `%` est un caractère spécial dans une chaîne de formatage, puisqu'il permet de définir un format Si on veut afficher le caractère `%` lui-même, il faut écrire `%%` dans la chaine de formatage, afin que la fonction `printf` sache qu'il ne s'agit pas d'un format :

```
printf("resultat des ventes : %d %%",87);
resultat des ventes : 87 %
```

Les **principaux** codes de formatage sont :

code	résultat affiché
`%d`	nombre entier relatif décimal
`%u`	nombre entier naturel décimal
`%o`	nombre entier naturel octal
`%x` et `%X`	nombre entier naturel hexadécimal
`%p`	nombre entier représentant une adresse (pointeur)
`%c`	caractère
`%f`	nombre réel décimal
`%e` et `%E`	nombre réel décimal en notation scientifique

Remarque : par défaut, un réel est affiché avec *6* chiffres après la virgule.

Il est possible de préciser, entre le `%` et la lettre du format, un **modificateur** de formatage précisant le type de la donnée à afficher :

donnée	codes concernés	option	résultat affiché
entier	d, i, o, x, X, u	aucune	int
		h	(unsigned) short
		l	long
réel	f, e, E	aucune	float
		l	double
		L	long double

En plaçant une valeur entière entre le `%` et la lettre du format, on peut préciser le nombre de chiffres **minimal** à afficher. Pour un réel, le point décimal compte comme un chiffre. Si le nombre à afficher ne contient pas assez de chiffres, les chiffes manquants sont remplacés par des caractères *espace*. Si on préfère remplir le vide avec des caractères *0*, il suffit de faire précéder la valeur minimale d'un zéro.

```
printf("entier:%4d reel:%9f",12,1.23456);
entier:  12 reel: 1.234560
printf("entier:%04d reel:%09f",12,1.23456);
entier:0012 reel:01.234560
```

On peut également fixer la précision, c'est-à-dire le nombre de chiffres **après** la virgule en faisant suivre le nombre précédent d'un point et d'une valeur entière. Si le nombre à afficher contient plus de chiffres, un arrondi est réalisé. Sinon, le nombre est complété avec des zéros :

```
printf("%.8f %.2f",1.23456,1.23456);
1.23456000 1.23
```

Exercice 3

Soient les variables réelles (utilisez des `double`) suivantes : `x1=1.2345`,`x2=123.45`, `x3=0.000012345`, `x4=1e-10` et `x5=-123.4568e15`. Affichez leurs valeurs avec `%f` et avec `%e`.

Exercice 4

Soient les variables réelles (`float`) suivantes : `x1=12.34567`, `x2=1.234567`, `x3=1234567`, `x4=123456.7`, `x5=0.1234567` et `x6=1234.567`. Utilisez `printf` pour obtenir l'affichage suivant :

```
     12.35         1.23
1234567.00    123456.70
      0.12      1234.57
```

Exercice 5

Écrivez un programme qui affiche la valeur 1234,5678 de type `float` avec les formats `%d`, `%f` et `%e`. Qu'observe-t-on pour `%d` ?

5 Entrées formatées

La fonction `scanf` permet de saisir des valeurs de manière formatée. Le principe est le même que pour `printf` : on utilise une chaîne de formatage, et une suite de paramètres.

```
scanf(format,adr1,...,adrn);
```

La différence est qu'ici, les paramètres sont des adresses. Ainsi, pour saisir un entier, on fera :

```
int i;
scanf("%d",&i);
```

L'opérateur `&` permet de préciser qu'on passe en paramètre non pas la variable `i`, mais son adresse. On peut saisir plusieurs valeurs à la fois, l'utilisateur devra les séparer par un retour chariot, un espace ou bien une tabulation :

```
int i,j,k;
scanf("%d%d%d",&i,&j,&k);
```

Cela signifie que par défaut, il n'est **pas possible** de lire le caractère *espace* avec `scanf`.

On utilise avec `scanf` les mêmes codes de formatages (`d`, `u`, `o`, `x`, `X`, `c`, `f`, `e`, `E`) et modificateurs (`h`, `l`, `L`) que pour `printf`. On peut préciser la **longueur maximale** de la donnée lue, en insérant une valeur avant le code de formatage (ou le modificateur). Par exemple, pour saisir un entier de 5 chiffres maximum :

```
int i;
scanf("%5d",&i);
```

Il est également possible de préciser que les valeurs lues doivent être séparées par des caractères particuliers. Par exemple, pour lire *3* valeurs séparées par des points-virgules :

```
int i,j,k;
scanf("%d;%d;%d",&i,&j,&k);
```

Remarque : `scanf` utilise un accès bufférisé au clavier, et ne consomme pas le dernier retour chariot. On a donc le même problème que pour `getchar`.

Exercice 6

Écrivez un programme qui saisit un entier avec `scanf` et affiche le triple de cette valeur.

Exercice 7

Écrivez un programme qui saisit une heure au format suivant : *heures:minutes:secondes*. Le programme doit afficher les valeurs saisies de la manière suivante (en respectant l'alignement) :

```
Entrez l'heure (hh:mm:ss) : 1:2:34
 1 heure(s)
 2 minute(s)
34 seconde(s)
```