- Introduction à l'algorithmique et au langage C -

Calculette

TP n°1

1re année ESIEA - Semestre 1

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

2016 - 2017

Avant propos

Dans le TD « Hello World », nous avons appris à mémoriser et à afficher des données. Nous allons voir maintenant comment faire des opérations sur ces dernières grâce aux opérateurs.

Nous apprendrons également à récupérer des données depuis le clavier, l'un de ces fameux flux d'entrée que nous avons décrit dans le TP « Initiation au Shell ».

Et enfin, armés de toutes ces connaissances, nous allons coder un petit programme qui permet de réaliser les actions basiques d'une calculatrice.

1 Valeur de retour

Avant de rentrer dans le vif du sujet, nous allons aborder une notion de base : la valeur de retour.

Une procédure est un bout de code qui réalise une tâche (qui devient atomique). Toute procédure peut renvoyer une valeur, nous lui donnons alors le nom de « *fonction* ». Cette valeur renvoyée peut corresponde par exemple au résultat de sa tâche, ou encore au statut de la tâche réalisée (succès, échec, etc.).

Par l'intermédiaire de la fonction principale, un programme peut lui aussi renvoyer son statut. D'où notre fameux « `return 0;` » que nous plaçons à la fin de chaque fonction `main`. Le zéro, conformément aux standards POSIX, indique que le programme s'est bien déroulé.

Cela permet notamment de chaîner les programmes comme nous l'avons vu dans le TD « Initiation au Shell ».

Bonnes habitudes

Il faut toujours tester le statut de l'action précédente avant de passer à la suite (qui dépend souvent de la bonne réussite de cette action).

2 Opérateurs de calcul, d'assignation et d'incrémentation

2.1 Faire des calculs

Les opérateurs de calculs ne devraient pas vous dépayser car ce sont les mêmes que ceux que vous utilisez régulièrement.

Table 1: Opérateurs de calcul.

Opérateur	Utilisation	Exemple
+	**Addition** : permet d'ajouter deux valeurs	a + b
−	**Soustraction** : permet de soustraire deux valeurs	a − b
*	**Multiplication** : permet de multiplier deux valeurs	a * b
/	**Division Euclidienne** : permet de diviser deux valeurs	a / b
%	**Modulo** : permet de calculer le reste de la division Euclidienne de deux valeurs	a % b

2.2 Mémoriser les résultats

Ces opérateurs de calcul renvoient tous le résultat de l'opération qu'ils ont effectuée. Cela nous permet par exemple d'assigner la valeur calculée à une variable au moyen de l'opérateur d'assignation « = » :

```
/* Declare */
int a;
int b;
int sum;

/* Assign */
a = 1;
b = 1;

/* Calculate and assign */
sum = a + b; // The '+' operator returns the result of the addition, here: 2
```

L'opérateur « *égale* » renvoie lui aussi une valeur de retour : la valeur qu'il assigne. Ainsi, nous pouvons écrire des affectations en cascade comme cela :

```
a = b = c = 1; // Equivalent to a = (b = (c = 1));
```

2.3 Accumulation

En plus de l'opérateur « = », il existe d'autres opérateurs d'assignation qui permettent d'affecter le résultat à la première variable concernée par le calcul. Nous les appelons également « opérateurs d'accumulation ».

Table 2: Opérateurs d'assignation.

Opérateur	Utilisation	Exemple	Equivalent
=	**Assignation** : affecte une valeur à une variable	var = a	-
+=	**Addition & Assignation** : additionne deux valeurs et affecte le résultat à la première variable	var += a	var = var + a
−=	**Soustraction & Assignation** : soustrait deux valeurs et affecte le résultat à la première variable	var −= a	var = var − a
*=	**Multiplication & Assignation** : multiplie deux valeurs et affecte le résultat à la première variable	var *= a	var = var * a

(Suite page suivante)

Opérateur	Utilisation	Exemple	Equivalent
/=	**Division Euclidienne & Assignation** : divise deux valeurs et affecte le résultat à la première variable	var /= a	var = var / a
%=	**Modulo & Assignation** : calcule le reste de la division Euclidienne de deux valeurs et affecte le résultat à la première variable	var %= a	var = var % a

```c
/* Declare */
int var1;
int var2;

/* Assign */
var1 = 1;
var2 = 1;

/* Calculate and assign */
var1 += var2; // Now var1 is equal to 2 (equivalent to var1 = var1 + var2)
```

2.4 Incrémentation

Quand nous voulons augmenter la valeur d'une variable de 1 (pour un compteur par exemple) plutôt que d'écrire : `var = var + 1` (qui est un peu verbeux), nous pouvons utiliser un opérateur spécial, dit d'incrémentation. Cela marche aussi pour la soustraction.

TABLE 3: Opérateurs d'incrémentation.

Opérateur	Utilisation	Exemple	Equivalent
++	**Incrémentation** : ajoute 1 à la valeur de la variable	$var++$	var = var + 1
−−	**Décrémentation** : retranche 1 à la valeur de la variable	$var--$	var = var − 1

Attention !
Ces deux opérateurs renvoient la valeur de la variable **avant** l'opération !

Par exemple :

```c
/* Declare */
int var;

/* Assign */
var = 1;

printf("%d\n", var++); // The operator returns the value 1 (and not 2)
printf("%d\n", var); // But var's value is now really 2
```

2.5 À vos codes !

EXERCICE 1

 Créez le fichier `operateurs.c` et ouvrez-le avec un éditeur de texte. Écrivez le code source minimal (*Hello World !*) d'un programme C. Sauvegardez-le et vérifiez qu'il compile bien.

EXERCICE 2

 Dans ce programme, ajoutez les instructions suivantes :

- Déclarez un entier `valI` puis un flottant `valF`.
- Initialisez `valI` à 2 et `valF` à `3.5`.
- Incrémentez `valI`.
- Affichez le contenu de `valI` et de `valF`.
- Décrémentez `valI`.
- Ajouter 3.0 à `valF`.
- Affichez le contenu de `valI` et de `valF`.
- Multipliez les deux variables par 2.
- Affichez le contenu de `valI` et de `valF`.
- Calculez le reste de la division euclidienne de 5 par 2 et le soustraire à `valI`
- Divisez `valF` par 7.
- Affichez le contenu de `valI` et de `valF`.

3 Embranchements et opérateurs de comparaison

Dans le TD « Déroulement d'un algorithme » nous avons découvert les structures de contrôles appelées embranchements. Nous allons donc voir maintenant comment les écrire en langage C. À part la syntaxe, nous retrouvons globalement la même chose :

Les embranchements de type SI vont s'écrire sous la forme :

```
if (TEST) {
   // Actions
}
```

Ceux de type SI-SINON :

```
if (TEST) {
   // Actions
} else {
   // Another actions
}
```

Et enfin les SI-SINON-SI :

```
if (TEST) {
   // Actions
} else {
   if (TEST) {
      // Another actions
   } else {
      // Another actions
   }
}
```

Nous avions vu aussi la boucle SELON, un peu différente de ces trois là, car nous l'utilisons quand nous intéressons aux multiples valeurs d'une variable donnée plutôt qu'à un état booléen. Voici comme l'écrire :

```
switch (VARIABLE) {
   case 1:
      // Actions for the case VARIABLE is equal to 1
      break;
   case 2:
      // Actions for the case VARIABLE is equal to 2
      break;
   default:
      // Actions for all the other case non described above
      break;
}
```

L'instruction break est très important car elle sert à sortir de la condition ! Si nous ne la mettons pas, nous considérerons le cas suivant quoi qu'il arrive. Or nous ne voulons qu'un et un seul cas en choix.

Revenons aux embranchements de type SI-SINON. Nous avons vu qu'ils doivent effectuer un test. Ainsi TEST peut être remplacé par n'importe qu'elle expression booléenne, c'est-à-dire qui est soit vraie (1), soit fausse (0).

Nous retrouvons à peu près les mêmes opérateurs de comparaisons :

TABLE 4: Opérateurs de comparaison.

Opérateur	Utilisation	Exemple
==	**Égalité** : renvoie 1 (VRAI) s'il y a égalité entre deux valeurs et 0 (FAUX) sinon	a == b
!=	**Différence** : renvoie 1 (VRAI) s'il n'y a pas égalité entre deux valeurs et 0 (FAUX) sinon	a != b
<	**Infériorité stricte** : renvoie 1 (VRAI) si une valeur est strictement inférieure à une autre et 0 (FAUX) sinon	a < b
<=	**Infériorité & Égalité** : renvoie 1 (VRAI) si une valeur est inférieure ou égale à une autre et 0 (FAUX) sinon	a <= b
>	**Supériorité stricte** : renvoie 1 (VRAI) si une valeur est strictement supérieure à une autre et 0 (FAUX) sinon	a > b
>=	**Supériorité & Égalité** : renvoie 1 (VRAI) si une valeur est supérieure ou égale à une autre et 0 (FAUX) sinon	a >= b

EXERCICE 3

Créez le fichier `estDivisible.c` et ouvrez-le avec un éditeur de texte. Écrivez le code source minimal (*Hello World !*) d'un programme C. Sauvegardez-le et vérifiez qu'il compile bien.

EXERCICE 4

— Déclarez un entier `var`.
— Initialisez-le avec un entier positif de votre choix.
— Écrivez le code qui permet d'afficher « Le nombre (var) est divisible par 2. » à condition que le contenu de var soit effectivement divisible par 2. Rappelez vous que A est divisible par B si et seulement si le reste de la division euclidienne de A par B vaut zéro.
— Écrivez des codes similaires pour vérifier la divisibilité de `var` par 3, 4, 5, 6, 7, 8 et 9. Veillez à bien tester à chaque étape.

EXERCICE 5

 Créez le fichier `estPair.c` et ouvrez-le avec un éditeur de texte. Écrivez le code source minimal (*Hello World !*) d'un programme C. Sauvegardez-le et vérifiez qu'il compile bien.

EXERCICE 6

 Écrivez un programme qui déclare un entier `var` initialisé à la valeur (positive) de votre choix. Puis qui affiche « (var) est paire » si `var` est paire, et « (var) impaire » sinon.

4 Une touche de logique !

Pour pouvoir réaliser des tests un peu plus complexes, rajoutons les opérateurs logiques.

Table 5: Opérateurs logiques.

Opérateur	Utilisation	Exemple
&&	**ET logique** : renvoie 1 (VRAI) ou 0 (FAUX) selon le résultat de l'évaluation logique de deux valeurs booléennes	a && b
||	**OU logique** : renvoie 1 (VRAI) ou 0 (FAUX) selon le résultat de l'évaluation logique de deux valeurs booléennes	a || b
!	**NON logique** : renvoie la négation logique de la valeur booléenne	!a

Pour rappel, voici les tables de logique :

Table ET

a	**b**	**a ET b**
0	0	0
0	1	0
1	0	0
1	1	1

Table OU

a	**b**	**a OU b**
0	0	0
0	1	1
1	0	1
1	1	1

Table NON

a	**NON a**
0	1
1	0

4.1 Un peu de lecture

QUESTION 1

 Qu'affiche ce programme ?

```
int var1;
int var2;

var1 = 5/2;
var2 = 0;

if(var1 == 2 && var2 != 0) {
   printf("Youpi !\n");
}

if(var1 == 1 || var1 == 2) {
   printf("Youpla !\n");
}

if(! (var1 != 2 || var2 == 0) ) {
   printf("Boum 1 !\n");
}

if(! (var1 != 1 && var2 != 2) ) {
   printf("Boum 2 !\n");
}

if(var1 >= 0 && var1 <= 5 ) {
   printf("La valeur de var est entre 0 et 5.\n");
}

if(var2 < 0 && var2 > 5 ) {
   printf("Mais comment est-ce possible ?\n");
}
```

5 Récupérer un flux d'entrée

Jusqu'à présent, lorsque nous avons initialisé une variable, nous avons mis sa valeur directement dans le code. Aussi, à chaque fois que nous avons modifié cette valeur, nous avons dû relancer la compilation. La fonction `scanf` va nous permettre de récupérer des valeurs saisies au clavier lors de l'exécution d'un programme, c'est-à-dire sans que nous ayons besoin de relancer la compilation.

Le mode de fonctionnement de `scanf` est très proche de celui de `printf` que nous avons vu dans le TD « *Hello World* », à la différence que nous devons lui indiquer dans quelles variables ranger les valeurs récupérées. Par exemple, si nous voulons récupérer un entier :

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

Nous retrouvons le descripteur de type que nous connaissons bien : `%d`, qui montre que nous attendons un entier au clavier.

Remarquez le caractère « & » devant le nom de la variable. Il permet d'indiquer à la fonction que nous nous intéressons à l'emplacement mémoire (dans lequel nous voulons ranger la nouvelle valeur) et non au contenu. Nous reviendrons sur cette notion un peu plus tard.

QUESTION 2

 Écrivez les lignes correspondantes pour les deux autres types de variable que vous connaissez.

Et si... ?

Mais que se passe-t-il si l'utilisateur se trompe au clavier ?

Avant d'utiliser la variable qui vient de recevoir sa nouvelle valeur, il est d'usage avec la fonction `scanf` de vérifier le statut qu'elle retourne. La fonction `scanf` retourne le nombre de valeurs qu'elle a récupérées, et 0 en cas d'erreur.

Nous allons donc effectuer un test sur cette valeur de retour. Si nous cherchons à récupérer un entier, alors `scanf` doit nous retourner 1 si tout s'est bien passé. Dans le cas contraire, nous affichons un message d'erreur et nous stoppons le programme avec un statut d'erreur (sinon son comportement risque de devenir très vite incertain).

```
int anInteger;
if(scanf("%d", &anInteger) != 1) {
   printf("Input error\n");
   return 1;
}
```

Règle fondamentale

Ne jamais faire confiance à rien ni personne en informatique ! Soyez parano :p !

6 Calculette

6.1 Objectif

Nous voulons réaliser un programme qui récupère en entrée un opérateur mathématique (addition, soustraction, multiplication, quotient et reste de la division euclidienne) et deux opérandes, et qui affiche le résultat de l'opération.

6.2 Calculette interactive

EXERCICE 7

— Commencez par créer le fichier `interactiveCalc.c`.
— Dans un premier temps nous allons demander à l'utilisateur d'entrer l'opérateur qu'il désire à l'aide d'un menu.

```
Please enter the operator:
1) add
2) substract
3) multiply
4) divide
5) modulo
```

On récupère donc un entier entre 1 et 5.
— puis nous demandons la valeur des opérandes (entières).
— En cas d'erreur de saisie, vous afficherez des messages d'erreur puis quitterez le programme.
— Sinon, à l'aide des embranchements nous réalisons le calcul demandé.
— Puis nous affichons le résultat. Par exemple :

```
12 + 4 = 16
```

— Compilez et testez.

6.3 Calculette fantôme

EXERCICE 8

— Commencez par créer le fichier `ghostCalc.c`.
— Copiez-collez le code précédent, puis commentez tous les `printf` (sauf les messages d'erreur).
— À la fin de votre programme, affichez uniquement la valeur du résultat.
— Compilez et testez.
— Que se passe-t-il si vous tapez la commande suivante ?

```
echo "1 12 4" | ./ghostCalc
```

6.4 Calculette flottante

EXERCICE 9

— Commencez par créer le fichier `floatCalc.c`.
— Copiez-collez le code précédent.
— Modifiez votre programme pour utiliser des opérandes à virgules flottantes (attention, un des opérateur disparait).
— Compilez et testez.

6.5 Calculette de pro

EXERCICE 10

— Commencez par créer le fichier `calc.c`.
— Copiez-collez le code précédent.
— Modifiez votre programme pour récupérer l'opérateur sous forme de caractère (`+`, `-`, `/`, `*`).
— Compilez et testez.
— Que faut-il taper pour tester votre programme ?

Mémo

« Il faut toujours tester les statuts. »