Licence d'informatique
Module de Pratique du C

Premiers travaux dirigés de pratique du C

Octobre 2004
révision de septembre 2006

1 Premiers abords du langage C

Ces premiers exercices abordent les notions d'identificateurs du langage, les valeurs entières, les expressions et les opérateurs simples.

Exercice 1 (Identificateurs)
Indiquer si les identificateurs suivants sont valides :

```
foo-1          _foo_bar       3cavaliers
foo.           tête           ___A_
_              a              while
```

(• ***Éléments de solution 1*** On peut rappeler qu'un identificateur est un mot utilisé (et bien choisi... des identificateurs courts pour les variables locales "anonymes" (`i`, `pos`), des noms parlant pour les variables globales (`surname_table`)) par le programmeur pour désigner une entité manipulé : une variable, une fonction...

- `foo-1` contient un opérateur et n'est donc pas valide ;
- `_foo_bar` est valide ; attention début par un _, habitude de les réserver pour cacher des identificateurs dans une bibliothèque...
- `3cavaliers` commence par un entier et n'est donc pas valide ;
- `foo.` contient un opérateur et n'est donc pas valide ;
- `tête` contient un caractère accentué. Peux être valide, mais n'est certainement pas portable. On se limite au codage ASCII et aux caractères sur 7 bits ;
- `___A_` est valide ; illisible (deux ou trois _ ,
- `_` (particulièrement illisible) et `a` sont valides ;
- `while` n'est pas valide, c'est un mot-clé. Les mots-clés sont réservés.

On peut ajouter que de manière générale, en C, on utilise des identificateurs en minuscule. On réserve les majuscules pour les valeurs constantes (`#define` ou `const`). Il est aussi habituel d'utiliser le trait bas `une_variable` plutôt que la capitalisation `uneVariable`. •)

Exercice 2 (Validité d'expressions)
On suppose les variables entières suivantes définies et affectées des valeurs indiquées :

```
A=20    B=5     C=-10   D=2     X=12    Y=15
```

Évaluez les expressions valides parmi les expressions suivantes :

```
5*X+2*3*B/4     A == B = 3      A += X + 2      A != C *= -D
A %= D++        A %= ++D        (X++) * A+C     A-A == B-B
!(X-D+C) || D   A && B || !0    C=12--          (1<<(2<<1))==((1<<2)<<1)
```

Corriger les expressions invalides en utilisant le parenthésage. On se référera au tableau 1 qui liste les opérateurs et leurs priorités.
(• ***Éléments de solution 2*** Un rappel des opérateurs est nécessaire ! On lit le tableau des opérateurs. L'ordre de priorité G pour l'opérateur $\times$ signifie que `g` $\times$ `c` $\times$ `d` est évalué de gauche à droite soit `(g` $\times$ `c)` $\times$ `d`.

- `5*X+2*3*B/4` est évalué de gauche à droite avec priorité de la multiplication sur l'addition, soit `(5*X)+(((2*3)*B)/4)`, soit `60+(30/4)`, le `/` dénote la division entière, on obtient donc `60+7`, soit `67`.

TAB. 1 – Opérateurs du langage C

16	`()  []  ->  .`	G
15	`++  --` (postfixé)	D
14	`!  ~  ++  --` (préfixé) `-` (unaire)	D
	`*` (indirection) `&` (adresse) `sizeof`	D
13	`*` (multiplication) `/  %`	G
12	`+  -`	G
11	`<<  >>`	G
10	`<  <=  >  >=`	G
9	`==  !=`	G
8	`&` (et bit à bit)	G
7	`^`	G
6	`|`	G
5	`&&`	G
4	`||`	G
3	`?:`	D
2	`=  +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=`	D
1	`,`	G

- `A == B = 3` on évalue le `==` (plus prioritaire), on obtient donc `0` (faux) et l'expression `0=3` qui est invalide : affectation dont la partie gauche n'est pas une *l-value*.
  On peut écrire `A == (B=3)` qui affecte B de la valeur `3` puis compare cette valeur affectée à `A` et retourne donc `0` (faux).
- `A += X + 2` il s'agit d'une affectation additive de `A` avec la valeur de l'expression `X+2`. `A` est donc affectée de `34` et cette valeur est retournée comme la valeur de l'expression.
- `A != C *= -D` est évalué en commençant par `-D`, ensuite `A != C` est évalué à une valeur non nulle (vrai). Puis erreur, affectation dont la partie gauche n'est pas une *l-value*.
- `A %= D++` affecte la valeur `A` de `20%2` soit `0` puis incrémente `D`, qui vaut maintenant `3`, retourne la velur de `A` soit `0`.
- `A %= ++D` incrémente `D` (qui vaut alors `3`) puis réalise l'affectation de `A` avec la valeur `20%3`, soit `2`. Retourne cette valeur `2`.
- `(X++) * A+C` évalue l'expression `X*A+C` c-a-d `(X*A)+C` soit 230, cette valeur est retournée alors que `X` est incrémenté à `13`.
- `A-A == B-B` les sous expressions arithmétiques sont évaluées (toutes deux à `0`) puis la comparaison est réalisée, retourne une valeur vrai (i.e. non nulle).
- `!(X-D+C) || D` L'expression arithmétique `X-D+C` est réalisée et vaut `0` (faux). On a donc `(!0)||D` soit une valeur non nulle (vrai).
- `A && B || !0` est évaluée `(A && B) || !0`, soit `(vrai && vrai) || !0`, soit vrai (une valeur non nulle).
- `C=12--` est invalide, la valeur `12` n'est pas une *l-value* et ne peut être décrémentée. On ne peut pas plus écrire `(C=12)--`.
- `(1<<(2<<1))==((1<<2)<<1)` est évalué `(1<<(4))==((4)<<1)`, soit `16==8`, soit `0` (faux). On reviendra sur la représentation binaire pur des entiers positifs...

En conclusion : inutile d'apprendre le tableau des opérateurs par cœur ; systématiquement parenthéser les expressions complexes (sauf peut être `*` et `+`) ; ne pas réaliser d'affectation au milieu d'une expression (sauf de rares incrémentations/décrémentations). •)

Exercice 3 (Typage entiers et conversion)
Soient les déclarations suivantes :

```
long int i = 15;
char c = 'A';
short int j = 10;
```

Évaluez et donnez le type des expressions valides suivantes :

```
c+1                    c+i                    c+j
3*c+2*c                2*c+(i+10)/j           2*c+(i+10.0)/j
j=(i+10.0)/j           c = 666                j *= 3.14
```

(• ***Éléments de solution 3***

1. On rappelle que la norme C définit les types d'entiers suivants :
   - `int` 4 octets, généralement ;
   - `short int` (ou `short` tout simplement) de taille $\leq$ `sizeof int`, souvent 2 octets ;
   - `long int` (ou `long`) de taille $\geq$ `sizeof int`, souvent 4 octets sur les machines 32 bits, 8 sur les machines 64 bits ;
   - `char` **1 octet, par définition** ; ce sont des valeurs entières.

   La norme impose aussi qu'un type non signé (`unsigned`) et un type signé de même nom possède la même taille.
2. Le codage des entiers non signés doit être un codage binaire pur. (De plus, la représentation d'une valeur signée positive et de la même valeur non signée doivent être identique.) Rien n'est dit sur le codage des valeurs négatives des entiers signés, les compilateurs utilisent souvent le complément à deux.
3. Les calculs sont fait sur des registres du processeur. En C, il n'y a pas de calcul sur des valeurs plus petite qu'un registre...

   On ne fait jamais de calcul sur les types plus petits qu'un `int`. Par exemple dans le cas `short + short`, on a deux conversions vers `int` puis une addition entière. Le résultat est un `int`. L'artitmétique sur les `short` est donc plus coûteuse que sur les `int`.
4. **Contagion** Une conversion vers la taille du plus grand des opérandes est réalisée si nécessaire pour chaque opérateur.
5. L'arithmétique sur les entiers se fait sans erreur de dépassement. Avec la représentation en complément à 2 des valeurs négatives, on obtient une arithmétique modulaire (taille de la représentation). D'autres comportements sont possibles.
6. **Troncature** L'affectation d'une valeur hors limite provoque une perte d'information...

On en vient à l'exercice :
- `c+1`, arithmétique entière, conversion de la valeur de `c` en un `int` (de valeur code ASCII de `A`, c-a-d `65`), le résultat est un `int` de valeur `65+1`, que l'on peut aussi considérer comme le code ASCII de `B`.
- `c+i` est évalué en convertissant `c` en un `long int` le résultat est un `long int` de valeur code ASCII de 'A' + 15, soit `80`.
- `c+j` idem `c+i` de type `int`.
- `3*c+2*c` est évalué `(3*c)+(2*c)`. Les multiplications sont licites sur les `char` qui, répétons le, sont des valeurs entières converties en `int`. Attention, le résultat déborde de la valeur d'un `char`, mais ce n'est pas un soucis ; ce résultat est un `int`.
- `2*c+(i+10)/j` Le seul « piège » est l'opérateur `/` de division entière. `2*c+(i+10)/j` = `((int)130+((long) 25)/(int) 10)` = `(long) 132 + 2` = `(long) 132`
- `2*c+(i+10.0)/j` ici la division va être la division flottante car un des opérandes et flottant. On obtient `(int) 130 + (15+10.0)/10` soit `((int)130) + 2.5` soit `(float)132.5`.
- `j=(i+10.0)/j` l'affectation tronque la valeur flottante `132.5` (question précédente) à la valeur `132`.
- `c = 666` La valeur `666` est hors limite pour un `char` (et le compilateur doit le dire). Si cette valeur était issue d'une expression, le résultat serait indéterminé (modulo si codage en complément à 2 des valeurs négatives).
- `j *= 3.14` `j` reçoit la valeur `31`.

•)

2 Premiers programmes en C

Ces exercices ont pour but de vous familiariser avec la syntaxe du langage C :
- les structures de contrôle : `if`, `while`, `for`...
- quelques fonctions d'entrée/sortie : `printf()`, `scanf()`, `getchar()`...
- la structure de base d'un programme
- le codage des entiers positifs en binaire.

Exercice 4 (Factorielle)
Question 4.1 Donner le code d'une fonction

```
unsigned int factorielle (unsigned int n) ;
```

qui calcule $n!$. On développera une version itérative et une version récursive de la fonction.
(• ***Éléments de solution 4.1***

- Présentation d'une fonction : la déclaration (aussi nommé prototype. Il existe une fonction qui...), la définition (le code ou corps de la fonction), le type de la valeur retournée par la fonction, les arguments ou paramètres...
- Le style (type sur une ligne, nom de la fonction sur une ligne, un paramètre par ligne, accolade { en fin de prédicat, accolade } seule sur une ligne, exception } `else` { ...). Ne pas négliger le style.
- On illustre les différentes structures de contrôle.
- `--n` et `n--`

```
unsigned int
factorielle_tantque (unsigned int n)
{
    unsigned int resu = 1;

    while (n>1)
        resu *= n--;
    return resu;
}
```

On arrive à ce résultat progressivement :

- on peut commencer par utiliser un variable `i` pour compter de `1` à `n` ;
- noter que le paramètre `n` est une variable locale modifiable localement dans la fonction, sans incidence sur la valeur de l'appelant ;
- on peut écrire dans un premier temps `resu = resu * i; i++;`

```
unsigned int
factorielle_jusqua (unsigned int n) /* n != 0 */
{
    unsigned int resu = 1;

    do
        resu *= n--;
    while (n>1);

    return resu;
}

unsigned int
factorielle_for (unsigned int n)
{
    int resu = 1, j;
    for (j=n; j>1; j--)
        resu *= j;
    return resu;
}

unsigned int
factorielle_recu (unsigned int n)
{
    if (n<=1)
        return 1;
    else
        return n * factorielle_recu (n-1);
}
```

On peut introduire le `case`, même plus naturel ici :

```
unsigned int
factorielle_recu_case (unsigned int n)
{
    switch n {
        case 0, 1 :
            return 1;
        default :
            return n * factorielle_recu_case (n-1);
    }
}
```

et même :

```
unsigned int
factorielle_recu_case2 (unsigned int n)
{
    switch n {
        case 0, 1 :
            return 1;
        case 2 :
            return 2;
        case 3 :
            return 6;
        ...
        default :
            return n * factorielle_recu (n-1);
}
```

On a aussi la possibilité d'introduire l'opérateur `?:` :

```
unsigned int
factorielle_recu_expr (unsigned int n)
{
    return n<= 1 ? 1 : n * factorielle_recu_expr (n-1);
}
```

•)

Question 4.2 Donner le code d'un programme qui lit un entier et affiche sa factorielle.
(• ***Éléments de solution 4.2***

- Les fonctions `scanf()` et `printf()`; `stdio.h`.
- La fonction `main()` (mais pas ses arguments). Elle retourne bien un entier (convention ; philosophie... !)
- `exit()` et `EXIT_SUCCESS` et `EXIT_FAILURE`.
- Les fonctions « locales » (`static`);

```
#include <stdlib.h>     /* pour exit(), EXIT_SUCCESS, et EXIT_FAILURE */
#include <stdio.h>      /* pour printf() et scanf() */

static unsigned int
factorielle (unsigned int n)
{
    ...
}

int                 /* par convention */
main ()
{
```

```
    int n, res;
    printf("Donner un entier :");
    scanf("%d", &n);
    res = factorielle(n);
    printf("Factorielle %d = %d\n", n, res);

    exit(EXIT_SUCCESS);
}
```

•)

Exercice 5 (Grande ligne)
Soit la fonction C `maxl_line()` qui lit un texte sur l'entrée standard et retourne la longueur de la plus grande ligne du texte.

On pourra utiliser la fonction `getchar()` de la bibliothèque standard qui renvoie le code ASCII du caractère lu sur `stdin`, la valeur `EOF` en fin de fichier ou cas d'erreur.

Question 5.1 Donnez le prototype de la fonction `maxl_line()`.
(• ***Éléments de solution 5.1***
- elle retourne une valeur entière, donc `int` a priori ; `unsigned int` vu l'info retournée, mais on est pas à un bit près, `int` est suffisant.
- pas de paramètre ; on écrit pas nécessairement `void`.

```
int maxl_line();
```

•)

Question 5.2 Donnez la définition de la fonction `maxl_line()`.
(• ***Éléments de solution 5.2***
- Il s'agit du corps de la fonction.
- Fin de ligne = le caractère `'\n'`
- Introduction de
  ```
  #include <stdio.h>

  int getchar (void);
  ```
- Pourquoi `c` est déclaré `int` et non `char`
- L'usage n'est pas d'écrire
  ```
  c = getchar()
  while (c != EOF) {
      ...
     c = getchar();
  }
  ```
  mais il faut faire attention aux `()` nécessaires.

```
int
maxl_line()
    int c;              /* le caractère courant */
    int longueur= 0;    /* la longueur de la ligne courante */
    int maximum= 0;     /* le maximum actuel */

    while ((c=getchar()) != EOF) {
        longueur++;
        if (c == '\n') {      /* fin de  ligne */
            if (longueur > maximum)
                maximum = longueur;
            longueur = 0;
        }
    }
}
```

Oui, on compte le caractère fin de ligne comme un caractère. On peut faire un autre choix... •)

Question 5.3 Donnez le code d'un programme qui lit son entrée standard et affiche la longueur de la plus longue ligne.

(• ***Éléments de solution 5.3***

– Il suffit de placer les `#include`, une fonction `main()`, et la fonction `maxl_line()` dans un fichier.

```
#include <stdlib.h>     /* pour exit(), EXIT_SUCCESS, et EXIT_FAILURE */
#include <stdio.h>      /* pour printf() et getchar() */

static int
maxl_line()
{
    ...
}

int                   /* par convention */
main ()
{
    int max;

    max = maxl_line();

    printf("La longueur de la plus longue ligne est %d\n", max);

    exit(EXIT_SUCCESS);
}
```

•)

Exercice 6 (Parenthésage d'une expression)

On lit sur l'entrée standard (`stdin`) un texte, terminé par `EOF`, dont on veut vérifier que le parenthésage est correct. On sortira sur la sortie standard (`stdout`) un message pour le résultat. De plus, la commande se terminera sur un succès si et seulement si l'expression est bien parenthésée.

(• ***Éléments de solution 6***

– Le `switch/case/break` n'avait pas été utilisé dans l'exercice précédent...

– Introduction de `exit()` en cas d'erreur.

```
#include <stdio.h>

int
main ()
{
  int ouvertes = 0;
  int carac;
  while ((carac=getchar()) != EOF)
  switch (carac) {
    case '(' :
      ouvertes ++;
      break;
    case ')' :
      if (ouvertes <= 0)
        printf("Erreur de fermeture\n");
        exit(EXIT_FAILURE);
      else
        ouvertes --;
        break;
      default : ;
  }
  if (ouvertes != 0) {
```

```
    printf ("Manque %d fermetures\n", ouvertes);
    exit(EXIT_FAILURE);
  }
  printf ("Parenthésage correct\n");
  exit(EXIT-SUCCESS) ;
 }
```

•)

Exercice 7 (Exemples de macros)
Définir à l'aide de macros `cpp` les « constantes » VRAI et FAUX.

Écrire ensuite, le plus simplement possible, une macro `cpp` qui teste si un caractère `c` passé en paramètre est un chiffre ou une lettre majuscule : `CHIFROUMAJ(c)`.
(• ***Éléments de solution 7***

- Faux c'est 0, le reste est vrai.
- Attention les tests du genre `expr == VRAI` doivent être simplement écrits `expr`

```
#define VRAI           (1==1)
#define FAUX           (1==2)

#define FAUX           0
#define VRAI           (! FAUX)
```

- macro parametrée, attention à la syntaxe (pas d'espace avant la `()`
- une macro peut en utiliser une autre
- notez que `isdigit()` et cie sont définies dans `<ctype.h>`

```
#define CHIFFRE(c)     ((c>='0')&&(c<='9'))
#define MAJUSCULE(c)   ((c>='A')&&(c<='Z'))
#define CHIFROUMAJ(c)  (CHIFFRE(c)||MAJUSCULE(c))
```

- attention à l'emploi multiple des paramètres des macros
- attention aux parenthèses autour de l'utilisation des paramètres

```
#define bogus_carre(x) x*x
bogus_carre(getint())  -> getint() * getint()
2 * bogus_carre(a+1)       -> 2 * a + 1 * a + 1 -> (2 * a) + (1 * a) + 1
#define carre(x)  ((x)*(x))
```

avec `getint()` qq chose du genre de `getchar()` •)

Exercice 8 (Word count)
Donnez un programme C qui compte le nombre de mots du texte fourni en entrée.
(• ***Éléments de solution 8***

- Pas de difficulté, si ce n'est sur la définition de "mot"...
- On peut introduire
  `#include <ctype.h>`

  `int isalpha (int c);`
  qui vérifie si une caractère est un caractère alphabétique, i.e une lettre.
- Vive les automates... et cela servira pour le TP pretty printer.

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

```
#define ETAT_IN_WORD            1    /* un enum serait preferable */
#define ETAT_OUT_WORD           2

int
main ()
{
    int wc = 0;                 /* word count */
    int c;
    int etat = ETAT_OUT_WORD;

    while ((c=getchar()) != EOF) {
        switch (etat) {
            case ETAT_OUT_WORD:
                if (isalpha(c)) {
                    /* on entre dans un mot, on le compte */
                    etat = ETAT_IN_WORD;
                    wc++;
                }
            case ETAT_IN_WORD:
                if (! isalpha(c)) {
                    /* on sort d'un mot */
                    etat = ETAT_OUT_WORD;
                }
        }
    }

    printf("%d mots\n", wc);

    exit(EXIT_SUCCESS);
}
```

•)

Exercice 9 (Conversion caractères/entiers)
On calcule la valeur d'un entier entré sur `stdin` sous forme d'une suite de caractères (terminée par un caractère autre qu'un chiffre).

L'arithmétique « classique » peut s'appliquer sur le type `char`. La valeur numérique d'un caractère est le code ASCII de ce caractère (65 pour `'A'`, 66 pour `'B'`...).

(• ***Éléments de solution 9***

– Arithmétique des caractères

```
unsigned int lire_entier ()
{
  unsigned int resu = 0 ;
  int carac = getchar() ;
  while (isdigit(carac)) {
    resu *= 10 ;
    resu += carac - '0' ;
    carac = getchar() ;
  }
  return resu ;
}
```

Ce qui peut être réécrit (au niveau du `getchar()`) :

```
unsigned int lire_entier ()
{
```

```
  unsigned int resu = 0 ;
  int carac ;
  while (carac = getchar (), isdigit(carac)) {
    resu *= 10 ;
    resu += carac - '0' ;
  }
  return resu ;
}
```

Attention à l'utilisation multiple d'un argument d'une macro, par exemple si on écrit

```
while (isdigit(carac=getchar()) {
```

•)

Exercice 10 (Format des entiers positifs)
On veut calculer le nombre de bits sur lequel sont codés les `unsigned int`. Le format des données n'est pas précisé dans les spécifications du langage C, il peut dépendre de la machine ou du compilateur.

L'opérateur arithmétique `x << lg` calcule un décalage sur `x` d'une longueur `lg` bits vers les bits de poids fort. Les `lg` bits de poids forts sont perdus, ceux de poids faible sont mis à 0.

On remarquera qu'on peut utiliser cet opérateur pour une multiplication (ou une division avec >>) par 2, 4..., à condition de faire attention aux pertes éventuelles d'informations.

(• ***Éléments de solution 10***
- On remarque que 0 est codé sous forme d'une suite de zéros et que 1 est codé sous la forme d'une suite de zéros suivie d'un unique 1.

```
int longint()
{
  unsigned int V = 1 ;
  int nb_decal = 0;
  while (V != 0) {
    V <<= 1 ;  /* ou  V = V << 1; */
    nb_decal ++ ;
  }
  return nb_decal ;
}
```

•)

Exercice 11 (Conversion décimal/binaire)
Définissez une fonction qui affiche sur la sortie standard la représentation binaire d'un entier non signé fourni en paramètre.

(• ***Éléments de solution 11***
- Déplacer un unique bit à 1 de la position la plus à gauche (poids fort) à la position la plus à droite (poids faible).
- `CHAR_BIT` défini dans `limits.h` donne le nombre de bits par octet (8 ?)
- on peut aussi utiliser le résultat de l'exercice précédent.
- introduit le constructeur `? :`, plus naturel qu'un `if` ici (c'est un point de vue...)
- on traite la position `bit`, du poids fort au poids faible. Pour chaque position on crée le masque nécessaire.

```
static void
binaire(unsigned int n)
{
    int bit;
```

```
    unsigned int mask;

    for (bit = ((sizeof(unsigned int) * CHAR_BIT) - 1);
         bit >= 0;
         bit--) {
            mask = (unsigned int) 1 << bit;
            putchar(n & mask ? '1' : '0');
    }
    putchar('\n');
}
```

– On peut se passer de la valeur de `bit` et manipuler directement `mask`.

```
static void
binaire(unsigned int n)
{
    unsigned int mask;

    printf("%d, 0x%X: ", n, n);

    for (mask = 1 << ((sizeof(unsigned int) * CHAR_BIT) - 1);
         mask > 0;
         mask >>= 1) {
            putchar(n & mask ? '1' : '0');
    }
    putchar('\n');
}
```

•)