Concepts avancés du langage C

par Jessee Michaël C. Edouard (Accueil)

Date de publication : 31 mars 2008

Dernière mise à jour : 09 janvier 2010

Ce tutoriel présente les concepts avancés du langage C (ANSI C). Une importance particulière a été accordée à sa bibliothèque standard, sans être toutefois exhaustif. En particulier, les fichiers seront étudiés dans un tutoriel à part.
Commentez cet article :

I - Compléments et rappels

I-A - Les différentes classes de variable

Une variable déclarée à l'intérieur d'un bloc est appelée une **variable locale**. Elle n'est connue qu'à l'intérieur de ce bloc. On dit que sa **visibilité** est limitée à ce bloc. Les arguments effectifs d'une fonction sont également des variables locales à cette fonction.
Une variable déclarée à l'extérieur de tout bloc (donc de toute fonction) est dite **globale**. Elle est donc visible de puis n'importe quelle fonction définie (dans le même fichier source !) après sa déclaration. L'utilisation des variables globales réduit la lisibilité du code. Ne les utilisez que lorsque cela est vraiment nécessaire.
Une variable globale est dite **permanente** (ou **statique**). Cela signifie que la mémoire utilisée par cette variable est valide pendant toute la durée d'exécution du programme. Elle ne perd donc jamais sa valeur.
A l'opposé les variables locales sont par défaut **automatiques**. Leur durée de vie est égale au temps d'exécution du bloc. On peut rendre une variable locale permanente à l'aide du mot-clé **static**. Par exemple :

```
#include <stdio.h>

void f(void);

int main()
{
    f();
    f();
    f();

    return 0;
}

void f()
{
    static int i = 100;

    printf("i = %d\n", i);
    i++;
}
```

Dans cet exemple, **i est une variable statique (permanente)** initialisée à 100. La mémoire utilisée par i est donc allouée avant même que le programme ne s'exécute et non à chaque appel de la fonction. Son contenu est initialisé à 100. Seulement, puisqu'elle a été déclarée dans f, elle ne sera **connue que dans f**. Mais i est tout sauf une variable « locale » (plus précisément : une variable automatique) à la fonction (f).
Au premier appel de f, on a donc i = 100 (puisqu'elle a été initialisée avec la valeur 100, avant même l'exécution du programme). Au deuxième appel, i = 101 et au troisième appel, i = 102.

I-B - Les expression constantes

Une **expression constante** est une expression dont la valeur peut être connue sans qu'on ait même à exécuter le programme. Par exemple les constantes littérales, l'adresse d'une variable statique, etc. sont des expressions constantes. Une expression constante peut également être composée, par exemple : 1 + 1.
Attention, une variable déclarée avec le qualificateur const est une variable, pas une expression constante !

I-C - Initialisation des variables

Le langage C permet d'initialiser une variable lors de sa déclaration.
Puisqu'une variable automatique n'est « activée » qu'au moment où l'on entre dans le bloc dans lequel elle a été déclarée, ce qui a lieu pendant l'exécution, on peut initialiser une variable automatique avec n'importe quelle expression et non forcément une expression constante.
De l'autre côté, puisque la mémoire utilisée par une variable statique est allouée avant même l'exécution, elle ne peut être initialisée qu'avec une expression constante.

Lorsqu'une variable statique est déclarée sans aucune initialisation, elle sera initialisée par le compilateur lui-même à 0. A l'opposé, le contenu d'une variable automatique déclarée sans initialisation est à priori indéterminé.

I-D - Les déclarations complexes

Nous avons déjà vu dans le tutoriel précédent comment créer de nouveaux types de données à l'aide du mot-clé **typedef**. On peut bien sûr exprimer un type sans l'avoir défini avec typedef. Pour obtenir le type d'un objet, il suffit de prendre sa déclaration, sans le point-virgule, et d'effacer l'identificateur. Par exemple, dans les déclarations suivantes :

```
int n;
char const * p;
double t[10];
```

n est de type int, p de type char const *, et t de type double [10] (« tableau de 10 double »).
Dans le troisième cas, double [10] est bien un type, cependant un tel type ne peut pas être utilisé dans une déclaration ou dans un typedef car les crochets doivent apparaître après l'identificateur. Par contre il pourra figurer en argument de l'opérateur sizeof ou dans le prototype d'une fonction par exemple.
Certains cas sont un peu plus délicats. Par exemple pour déclarer un tableau de 10 char *, on écrira :

```
char * t[10];
```

t est donc de type char * [10].
Or, comme nous le savons déjà, un tableau peut être implicitement converti en un pointeur vers son premier élément. Donc dans l'exemple ci-dessus, t (qui est un tableau de char *) peut être implicitement converti en un pointeur sur char * soit char **.
Dans un cas tel que :

```
int t[5][10];
```

- t est de type : int [5][10].
- t[0], t[1], ... t[4] sont de type : int [10].
- t est donc, pour un pointeur, un pointeur sur int [10].

Si l'on devait déclarer une variable p de ce type, on raisonnerait sans doute comme suit :

- p est un pointeur sur un int [10]
- *p est donc un int [10]

d'où :

```
int (*p)[10];
```

p est donc de type int (*)[10] !
Maintenant, en faisant :

```
p = t;
```

p pointe sur t[0] (qui est un tableau de 10 int). De même, p + 1 pointe sur t[1], p + 2 sur t[2] etc.
Evidemment, on peut aussi pointer sur t avec un simple int * (puisqu'un tableau, quelle que soit sa dimension, est un groupement d'objets de même type) mais cela nécessite cette fois-ci un cast et un peu plus d'habileté dans le calcul d'adresses bref, rien qui soit nouveau pour nous. C'est d'ailleurs la seule solution lorsqu'on veut passer un tableau à plusieurs dimensions à une fonction qui se veut d'être générique (en réalité, on utilisera plutôt un pointeur générique ...).

II - Fonctions

II-A - Pointer sur une fonction

Un pointeur rappelons-le est une variable destinée à contenir une adresse. Une adresse référence un objet en mémoire. Une fonction (pendant l'exécution) est tout simplement une suite d'instructions en mémoire. On peut donc pointer sur une fonction tout comme on peut pointer sur n'importe quelle variable.
Une fonction possède un type. Par exemple, considérons les fonctions f et g suivantes :

```
int f()
{
    printf("Fonction : f.\n");
    return 0;
}

int g()
{
    printf("Fonction : g.\n");
    printf("-------------\n");
    return 0;
}
```

Ces fonctions ont le même type et leur type est fonction retournant un int et ne prenant aucun argument.
Mais attention ! La taille occupée en mémoire par une fonction ne peut pas se déduire de son type. Par exemple, f occupe moins de place que g, cela est plus qu'évident. En conséquence : on ne peut pas affecter une « fonction » à une variable. Cependant, rien ne nous empêche de pointer sur une fonction. En se basant toujours sur l'exemple précédent, on peut déclarer le type de f et g comme suit :

```
typedef int F(void);
```

Et on peut donc déclarer un pointeur sur F de la manière suivante :

```
F * p;
```

Ensuite, il suffit de savoir que lorsqu'elle n'est pas utilisée en unique argument de sizeof ou de l'opérateur & ("adresse de"), une expression de type fonction est toujours convertie par le compilateur en l'adresse de cette fonction. Cela signifie que les expressions :

```
&f      f     *f    **f   ***f    ...
```

sont toutes équivalentes. Elles vallent toutes &f.
Donc une fois p correctement initialisée, il y a plus d'une manière d'appeler la fonction pointée :

```
p()    (p)()   (*p)()    ...
```

II-B - Point d'entrée d'un programme C

Au niveau global, un programme C est constitué d'une ou plusieurs fonctions, chacune ayant éventuellement une tâche bien précise. Quel que soit le nombre de fonctions constituant le programme, une doit être désignée comme étant le **point d'entrée** du programme, c'est-à-dire celle qui sera automatiquement appelée par l'environnement d'exécution lorsqu'on exécute le programme. Cette fonction est bien entendu dépendante de la plateforme. Par exemple sous Windows le nom de cette fonction est main pour une application console, WinMain pour une application fenêtrée, NtProcessStartup pour un driver, etc.
Cependant, pour permettre aux programmeurs d'écrire des applications totalement portables, le comité chargé de normalisation du langage C a désigné une fonction appelée **main** (héritée d'UNIX) comme point d'entrée d'un programme C standard. C'est ensuite au compilateur et au linkeur de faire en sorte que le programme puisse être exécutable sur la plateforme cible.

main est une fonction qui doit retourner un int. Les deux prototypes possibles pour cette fonction sont :

```
>> int main(void)
>> int main(int argc, char * argv[])
```

Ces arguments (argc et argv), lorsqu'ils sont utilisés, seront évidemment initialisés par l'environnement d'exécution (sinon par qui alors ?). Il est donc clair que les valeurs placées dans ces variables sont dépendantes de l'environnement. Néanmoins :
argv[0] n'est jamais NULL et pointe sur une chaîne qui représente le nom du programme. Si argc est supérieur à 1, argv[1] à argv[argc-1] représentent les **paramètres du programme**. Dans tous les cas, argv[argc] vaut toujours NULL. Dans la plupart des implémentations, les « paramètres du programme » sont les arguments de la ligne de commande. Par exemple sous Windows, en exécutant le programme prog.exe avec la ligne de commande :

```
prog abc xy
```

On aura :

- argc = 3
- argv[0] = "prog"
- argv[1] = "abc"
- argv[2] = "xy"
- argv[3] = NULL

II-C - Quelques fonctions utiles

La fonction exit

```
void exit(int status);
```

déclarée dans **stdlib.h**, permet de provoquer la terminaison normale du programme. Cette fonction reçoit en argument un entier qui sera retourné comme étant le code d'erreur de l'application. Les valeurs qu'on peut utiliser sont EXIT_SUCCESS (qui est généralement défini à 0), pour indiquer un succès et EXIT_FAILURE, pour indiquer un échec. L'utilisation de toute valeur autre que 0, EXIT_SUCCESS ou EXIT_FAILURE entraîne un comportement dépendant de l'implémentation. Attention ! Il faut avoir appelé free suite à un malloc avant d'appeler cette fonction, sinon la mémoire ne sera pas libérée ! En règle général : libérer toutes les ressources avant d'appeler exit.

La fonction system

```
int system(char const * string);
```

déclarée dans **stdlib.h**, permet de passer la chaîne string à l'interpréteur de commandes en vue d'être exécutée. L'utilisation de cette fonction rend donc un programme dépendant d'une certaine plateforme (les commandes Windows par exemple ne sont pas à priori les mêmes que les commandes UNIX ...).

La fonction sprintf

```
int sprintf(char * buffer, const char * format, ...);
```

déclarée dans **stdio.h**, est équivalente à fprintf sauf que la sortie est placée dans buffer plutôt que d'être écrite sur un fichier. Le programmeur doit être sûr que buffer est assez large pour pouvoir contenir la chaîne formatée avant d'utiliser cette fonction.

II-D - La récursivité

Un **algorithme récursif** est un algorithme dont la définition fait référence à lui-même. Par exemple, étudions l'algorithme qui permet de calculer la puissance d'un nombre c'est-à-dire an. Nous-nous limiterons au cas où a est de type réel et n de type entier relatif. Voici l'algorithme :

```
Algorithme « calcul de a p (n) »

/* Cet algorithme donne une définition de a p (n) (a puissance n) */

Debut
    Si n > 0
        a p (n) = a * (a p (n - 1))
    Sinon
        Si n < 0
            a p (n) = (1/a) p (-n)
        Sinon
            /* n == 0 */
            a p (n) = 1
        FinSi
    FinSi
Fin
```

Et voici un exemple d'implémentation en langage C :

```
double a_puissance_n(double a, int n)
{
    if (n > 0)
        return a * a_puissance_n(a, n - 1);
    else
        if (n < 0)
            return a_puissance_n(1/a, -n);
        else
            return 1.0;
}
```

Ainsi, en calculant a_puissance_n(1.5, 3) par exemple, on obtiendra successivement

- 1.5 * a_puissance_n(1.5, 2)
- 1.5 * 1.5 * a_puissance_n(1.5, 1)
- 1.5 * 1.5 * 1.5 * a_puissance_n(1.5, 0)
- 1.5 * 1.5 * 1.5 * 1.0

Nous pouvons bien constater que les appels de fonction récursive s'empilent. Une fonction reste toujours « en vie » tant que la suivante n'a pas retourné car elle doit justement attendre la valeur de retour de cette dernière avant de pouvoir à son tour retourner la sienne. Donc, derrière leur simplicité enfantine, les fonctions récursives consomment généralement énormément de mémoire. Heureusement, il n'y a pas d'algorithme récursif qu'on ne puisse écrire sous forme d'itération ...

III - Manipulation des flottants

III-A - Les nombres flottants

L'opérateur == est défini pour les nombres réels cependant, étant donné que cet opérateur ne fait qu'une comparaison bit à bit des valeurs comparées, il ne doit être utilisé que pour ce type de comparaison or ce n'est généralement pas ce que l'on souhaite faire. Pour tester l'égalité de deux réels, il faut calculer leur différence et voir si elle est assez petite (et assez petite est à définir par le programmeur !). En effet, les calculs avec les nombres réels ne se font pas toujours avec une précision infinie. Cela signifie qu'un calcul qui devrait donner pour résultat 0.0 par exemple peut ne pas retourner exactement 0.0 à cause des éventuelles pertes en précision qui ont pu avoir lieu lors des calculs

intermédiaires et c'est pourquoi, le programmeur doit toujours lui-même définir une certaine tolérance à l'erreur à chaque calcul utilisant les nombres réels.
La source de ce problème, c'est qu'il existe une infinité de nombres réels et l'ordinateur n'est bien entendu pas capable de pouvoir les représenter tous. En d'autres termes, l'ensemble des nombres représentables par l'ordinateur est discret et fini et non continu. La distance entre deux réels consécutifs de cet ensemble n'est cependant pas forcément constante. Il existe une macro définie dans float.h appelée DBL_EPSILON qui représente la plus petite valeur x de type double tel que 1.0 + x != 1.0. En d'autres termes, DBL_EPSILON est le réel tel que le successeur de 1.0 dans l'ensemble des nombres représentables par le système est 1.0 + DBL_EPSILON.
Donc comment choisir epsilon ? Tout dépend de l'ordre de grandeur des valeurs comparées et de la tolérance à l'erreur que l'on veut avoir. Il n'y a donc rien d'absolu (ou "d'universel") que l'on peut affirmer sur ce sujet. Si les nombres comparés sont de l'ordre de l'unité (mais pas trop proches de zéro), on peut choisir un epsilon absolu. Si la distance entre les deux nombres (c'est-à-dire |x - y|) est inférieure à epsilon, alors on doit les considérer comme "égaux". Dans les autres cas, il faut généralement être beaucoup moins tolérant (comparaison de nombres proches de zéro par exemple) ou beaucoup plus tolérant (comparaison de grands nombres). Il faut par exemple considerer un epsilon e tel que x est égal à y (on suppose que |x| < |y|) s'il existe un réel k tel que |k| < e et y = x + k.x. e est appelée erreur relative maximale autorisée. k est l'erreur relative qu'il y a entre x et y. L'avantage de cette méthode, c'est qu'elle choisit automatiquement le epsilon absolu qui va le mieux en fonction des nombres comparés. C'est donc cette méthode qui est utilisée dans la plupart des applications.
Voici une fonction qui permet de comparer deux réels avec deux epsilons spécifiés : le premier absolu et le deuxième relatif. Choisir 0 comme epsilon absolu pour l'ignorer (c'est-à-dire, comparer directement en utilisant le epsilon relatif).

```
/* Fonction dbl_cmp : Compare deux reels x et y */
/* Retourne :                                   */
/*    -1 si x < y                               */
/*     0 si x = y                               */
/*     1 si x > y                               */

int dbl_cmp(double x, double y, double eps_a, double eps_r)
{
    double a, b;
    int ret;

    /* On tente tout d'abord une comparaison bit a bit. */

    if (x < y)
    {
        a = x;
        b = y;
        ret = -1;
    }
    else if (x > y)
    {
        a = y;
        b = x;
        ret = 1;
    }
    else
        ret = 0;

    /* Si x != y, on tente alors une comparaison avec tolerance a l'erreur. */

    if (ret != 0)
    {
        /* Si eps_a != 0, l'utiliser. */

        if (b - a < eps_a)
        {
            ret = 0;
        }

        if (ret != 0)
        {
            /* Si on a encore ret != 0 (i.e. x != y), utiliser eps_r. */

            if ((b - a) / a < eps_r)
                ret = 0;
        }
```

```
    }

    return ret;
}
```

III-B - Les fonctions mathématiques

Les fonctions mathématiques sont principalement déclarées dans **math.h**. Le type flottant utilisé par ces fonctions est le type double.

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x); /* ArcSin(x) */
double acos(double x);
double atan(double x);
double atan2(double y, double x); /* atan(y/x), définie même si x = 0 */
double exp(double x);
double log(double x); /* ln(x) */
double log10(double x);
double sqrt(double x); /* racine carre (x) */
double pow(double x, double y); /* x puissance y */
double sinh(double x); /* sinus hyperbolique (x) */
double cosh(double x);
double tanh(double x);
double floor(double x); /* E(x) (partie entière (x)) */
double ceil(double x); /* E(x) + 1 */
double fabs(double x); /* | x | */
double fmod(double x, double y); /* x - n*y, | n*y | <= | x | < | (n+1)*y | */
```

IV - Les opérateurs de manipulation de bits

IV-A - Présentation

Ces opérateurs permettent d'effectuer des opérations bit à bit sur une expression de type entier.

IV-B - L'opérateur ET (AND)

a & b = 1 si et seulement si a = 1 **et** b = 1. Ex :

```
  00110101
& 01010011
  --------
  00010001
```

IV-C - L'opérateur OU (OR)

a | b = 1 dès que a = 1 **ou** b = 1. Ex :

```
  00110101
| 01010011
  --------
  01110111
```

IV-D - L'opérateur OU EXCLUSIF (XOR)

a ^ b = 1 si (a = 1 et b = 0) **ou** (a = 0 et b = 1). Ex :

```
  00110101
^ 01010011
  --------
  01100110
```

IV-E - L'opérateur NON (NOT)

~a = 1 si a = 0 et 0 si a = 1. Ex:

```
~00110101 = 11001010
```

IV-F - L'opérateur de décalage vers la gauche (<<)

```
11000011 << 1 = 10000110 /* décalage de 1 bit vers la gauche */
```

IV-G - L'opérateur de décalage vers la droite (>>)

Attention ! Le résultat d'une opération de décalage vers la droite sur un entier signé est dépendant de l'implémentation (à cause du bit de signe ...).

```
11001111 >> 2 = 0011001111 /* décalage de 2 bits vers la droite */
```

IV-H - Autres opérateurs

On a aussi les opérateurs &=, |=, ^=, <<= et >>=

V - Structures, unions et énumérations

V-A - Les structures

Une **structure** est une variable qui regroupe une ou plusieurs variables, non nécessairement de même type (contrairement aux tableaux), appelées champs. L'accès à un champ d'une structure se fait par l'intermédiaire de l'opérateur point. Par exemple :

```
struct {char nom[20]; int age;} marc, luc, jean;
...
strcpy(marc.nom, "Marc");
marc.age = 24 ;
...
```

D'habitude on renomme tout d'abord une structure avant de l'utiliser. Par exemple :

```
typedef struct {char nom[20]; int age;} PERSONNE;
...
PERSONNE marc, luc, jean;
...
```

Une structure peut être également nommée. Par exemple :

```
struct personne_s {char nom[20]; int age;} marc, luc, jean;
```

Dans ce cas, marc, luc et jean sont de type struct personne_s. On aurait également pu écrire :

```
struct personne_s {
```

```
    char nom[20];
    int age;
};
```

Puis :

```
struct personne_s marc, luc, jean;
```

Et bien entendu, du moment qu'une structure a été nommée, on peut très bien écrire :

```
typedef struct personne_s PERSONNE;
```

L'opérateur = (affectation) peut être utilisé avec des structures (pour copier une structure) mais dans la pratique on n'en a rarement besoin. En effet, puisque l'espace occupée en mémoire par une structure peut être gigantesque, copier une structure vers une autre pénalisera considérablement, non seulement en termes de performances mais aussi en termes de vitesse d'exécution (à cause du temps d'accès à une cellule mémoire). C'est pourquoi on utilise généralement des pointeurs pour manipuler les structures. L'opérateur -> permet d'accéder à un champ d'une structure via un pointeur. Par exemple :

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

struct personne_s {
    char nom[20];
    int age;
};

void affiche_personne(struct personne_s * p);

int main()
{
    struct personne_s jean;

    strcpy(jean.nom, "Jean");
    jean.age = 24;

    affiche_personne(&jean);

    return 0;
}

void affiche_personne(struct personne_s * p)
{
    printf("Nom : %s\n", p->nom);
    printf("Age : %d\n", p->age);
}
```

L'initialisation d'une structure se fait de la même manière que pour un tableau, avec les mêmes règles. Par exemple :

```
struct personne_s jean = {"Jean", 24};
```

V-B - Les unions

Une **union** est, en première approximation, une variable dont le type n'est pas figé (c'est-àdire une variable « caméléon »). En fait, une variable de type union est une variable composée de **champs** de types différents et non nécessairement de même taille, mais à la différence d'une structure, **les champs d'une union partagent le même emplacement mémoire**. Les unions permettent donc de faire une économie de mémoire. Du point de vue syntaxique, elles ressemblent beaucoup aux structures, il suffit de remplacer struct par union.
Le programme suivant montre un exemple d'utilisation de cette union, u, entre trois variables de types différents (n, x et p).

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

union trois_en_un {
    int n;
    double x;
    char * p;
};

int main()
{
    union trois_en_un u;

    printf("Entrez un entier : ");
    scanf("%d", &(u.n));
    printf("Vous avez entre : %d\n\n", u.n);

    printf("Entrez un nombre flottant : ");
    scanf("%lf", &(u.x));
    printf("Vous avez entre : %f\n\n", u.x);

    u.p = malloc(20);
    if (u.p != NULL)
    {
        strcpy(u.p, "Au revoir.");
        printf("%s\n\n", u.p);
        free(u.p);
    }

    return 0;
}
```

Puisque les champs d'une union utilisent le même espace mémoire, les unions sont également utilisées pour interpréter un même motif binaire dans différents contextes (par exemple pour interpréter le contenu d'une mémoire comme un flottant, ou comme un entier, ...), mais cette pratique est généralement utilisée dans les programmes de très bas niveau plutôt que dans des programmes portables.

V-C - Les champs de bits

Le langage C permet de créer, à l'intérieur d'une structure, des données (champs) dont la taille est spécifiée en **bits**. Ces champs doivent être de type entier. Par exemple :

```c
struct {
    unsigned a : 1;
    unsigned b : 1;
    unsigned c : 4;
    unsigned d : 3;
} x;
```

Ici, on a une structure x composée de 4 champs tous de type unsigned int : a de taille = 1 bit, b de taille = 1 bit, c de taille = 4 bits, et d de taille = 3 bits. Cela ne signifie pas que la taille de x est de 9 bits mais au moins 9 bits. La taille d'une donnée est toujours un multiple entier d'un "octet".
Les champs de bits peuvent être également utilisés, en faisant une union avec un autre type, dans des programmes dépendantes d'une plateforme particulière pour accéder directement aux bits d'une donnée quelconque. On retrouve alors l'intérêt de créer des champs sans noms : ils sont juste utilisés pour espacer des champs entre eux. Dans des programmes portables, on utilisera plutôt les opérateurs de manipulation de bits que les unions et les champs de bits.
Attention ! Puisque les champs d'un champ de bits représentent des bits d'une variable et non des variables indépendantes, on ne peut pas leur appliquer l'opérateur & (adresse de). Mais il peut bien entendu être appliqué à la structure elle-même qui est une variable comme toutes les autres.

V-D - Les énumérations

Une **énumération** est un sous-ensemble énuméré du type int. La déclaration d'une énumération ressemble un peu à celle d'une structure Par exemple :

```
typedef enum {
    PION,
    CAVALIER,
    FOU,
    TOUR,
    REINE,
    ROI
} CLASSE;
...
CLASSE classe = FOU; /* Ou egalement int classe = FOU; */
...
```

Qu'on aurait également pu écrire :

```
#define PION 0
#define CAVALIER PION + 1
#define FOU CAVALIER + 1
#define TOUR FOU + 1
#define REINE TOUR + 1
#define ROI REINE + 1
...
int classe = FOU;
...
```

L'opérateur = peut également être utilisé dans une déclaration d'un type énuméré pour assigner explicitement une valeur à une constante.

VI - Bibliothèque standard

VI-A - Fonctions à arguments en nombre variable

Le langage C permet de créer des fonctions dont le nombre d'arguments peut varier d'un appel à un autre c'est-à-dire n'est pas fixe. Nous avons déjà eu jusqu'ici l'occasion d'utiliser de telles fonctions comme par exemple printf et scanf. Maintenant nous allons voir comment en créer.
Une fonction qui accepte un nombre variable d'arguments doit au moins posséder un argument fixe, en effet même si la fonction peut recevoir un nombre variable d'arguments, elle doit quand même savoir à chaque appel combien d'arguments ont été passés. Par exemple, les fonctions printf et scanf comptent le nombre de spécificateurs de formats pour connaître le nombre d'arguments passés.
La bibliothèque standard du langage C est fournie avec des macros permettant d'accéder aux arguments « variables » de manière portable. Ces macros sont définies dans le fichier d'entête **stdarg.h**. Le principe est simple : à l'intérieur de la fonction, on crée une **liste d'arguments** (de type **va_list**). On initialise cette liste en pointant sur le dernier argument fixe à l'aide de la macro **va_start**. Puis on parcoure la liste à l'aide de la macro **va_arg**. Lorsqu'on en a terminé avec la liste, il faut appeler **va_end**.
Nous allons écrire une fonction qui calcule la moyenne des nombres passés en arguments. Le premier argument servira à préciser le nombre d'arguments variables passés.

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

double moyenne(int N, ...);

int main()
{
    printf("moyenne = %f\n", moyenne(4, 10.0, 20.0, 30.0, 40.0));
```

```
    return 0;
}

double moyenne(int N, ...)
{
    double moy = 0.0;

    if (N > 0)
    {
        va_list args;
        int i;
        double xi, somme = 0.0;

        va_start(args, N);
        for(i = 0; i < N; i++)
        {
            xi = va_arg(args, double);
            somme += xi;
        }

        va_end(args);
        moy = somme / N;
    }

    return moy;
}
```

VI-B - Les paramètres régionaux

Les **paramètres régionaux** sont un ensemble de paramètres qui peuvent changer selon la localisation tels que le symbole décimal, le symbole monétaire, le symbole de groupement des chiffres, etc. Les fonctions, macros et types ayant un lien avec ces paramètres sont déclarés/définis dans le fichier **locale.h**. La fonction **setlocale** :

```
char * setlocale(int category, const char * locale);
```

permet de modifier les paramètres régionaux utilisés par le programme. Cette fonction retourne NULL en cas d'erreur. Par défaut, tous les programmes utilisent les paramètres standard du langage C, c'est-à-dire comme si :

```
setlocale(LC_ALL, "C");
```

avait été appelé.
Les valeurs utilisables pour l'argument category sont LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, plus d'autres éventuelles macros définies par l'implémentation. Pour l'argument locale, "C" indique d'utiliser les paramètres standard du C et "" les paramètres de l'environnement d'exécution. L'implémentation peut évidemment autoriser d'autres valeurs. La fonction **localeconv** :

```
struct lconv * localeconv();
```

permet de récupérer les valeurs des paramètres courants. Par exemple :

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

int main()
{
    struct lconv * plconv;

    if (setlocale(LC_MONETARY, "") != NULL)
    {
        plconv = localeconv();
        printf("Le symbole monetaire est : %s\n", plconv->currency_symbol);
    }
```

```
    return 0;
}
```

VI-C - « Type » des caractères

Dans le fichier **ctype.h** sont déclarées des fonctions (ou macros) facilitant la manipulation des caractères. Ces fonctions (macros) attendent en argument un int puisqu'elles doivent aussi tester EOF. Leur comportement, à l'exception de isdigit et isxdigit, est dépendant de la valeur du jeu de paramètres LC_CTYPE.
Les fonctions suivantes permettent de tester le « type » d'un caractère et retourne VRAI (c'est-à-dire une valeur non nulle) lorsque le prédicat est vrai.

```
int isalpha(int c); /* c est une letrre */
int isdigit(int c); /* c est un chiffre decimal */
int isxdigit(int c); /* c est un chiffre hexadecimal */
int isalnum(int c); /* c est une lettre ou un chiffre decimal */
int islower(int c); /* c est une lettre minuscule */
int isupper(int c); /* c est une lettre majudcule */
int isspace(int c); /* c est un caractere d'espacement */
int ispunct(int c); /* c est une ponctuation */
int iscntrl(int c); /* c est un caractere de controle (DEL, ECHAP, ...) */
int isprint(int c); /* c est un caractere imprimable */
int isgraph(int c); /* isprint(c) && !isspace(c) */
```

Les fonctions toupper et tolower permettent de convertir, lorsque cela est possible, une lettre en majuscule respectivement en minuscule. Par exemple toupper('a') retourne 'A'.

VI-D - Les caractères larges

A la différence de la plupart des langages de haut niveau, le langage C traite les caractères comme de simples entiers (en fait un caractère **est** un entier) exactement de la même manière que le fait le processeur lui-même. Le type char n'est pas le seul type utilisé pour manipuler des caractères. Un **caractère large** (**wide character**) est un caractère plus large, en termes de taille, qu'un simple char, c'est-à-dire pouvant prendre plus de valeurs que ces derniers.
Une constante littérale de type caractère large s'écrit comme une constante de type char mais avec un préfixe L, par exemple : L'x'. De même, une constante chaîne constituée de caractères larges s'écrit comme une simple constante chaîne constituée de char mais avec le préfixe L, par exemple : L"Bonjour". Le standard définit un caractère large comme étant de type **wchar_t**, défini dans le fichier **stddef.h**.
Les fonctions impliquant des caractères ou chaînes de caractères en argument ou en valeur de retour ont donc toujours, à quelques très rares exceptions près, leur équivalent en wchar_t principalement déclarées dans **wchar.h** et **wctype.h**. Ainsi on a putchar/putwchar, getchar/getwchar, isalpha/iswalpha, toupper/towupper, strlen/wcslen, strcpy/wcscpy, printf/wprintf, scanf/wscanf, fgets/fgetws, etc. Par exemple :

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

int main()
{
    char c = 'a', *s = "Bonjour";
    wchar_t wc = 'a', *ws = L"Bonjour";

    printf("Voici un caractere simple : ");
    putchar(c);

    printf("\nEt voici un caractere large : ");
    putwchar(wc);

    printf("\n\n");

    printf("Voici une chaine de caracteres simples : %s\n", s);
    wprintf(L"Et voici une chaine de caracteres larges : %s\n", ws);

    return 0;
}
```

Un **caractère multioctet** (**multibyte character**) est un élément du jeu de caractères du système hôte ou, en termes plus simples, un ... caractère ! (Mais non nécessairement représentable par un char). On l'appelle multioctet puisqu'il peut être codé sur un (pourquoi pas ?) ou plusieurs octets. Un caractère multioctet peut donc être représenté par un char ou un tableau de char selon le cas. De même, une chaîne de caractères multioctets peut être représentée à l'aide d'un simple tableau de char. L'interprétation d'une chaîne de caractères multioctets est donc bien évidemment très dépendante du codage utilisé et ne rentre pas dans le cadre de ce tutoriel.
D'autre part, en fait, le type char a plutôt été désigné non seulement pour représenter un « octet » mais aussi n'importe caractère supporté par le langage lui-même (l'ensemble de ces caractères formant ce qu'on appelle le **jeu de caractères de base**), qui doit absolument tenir sur un octet, et non n'importe quel caractère du jeu de caractères du système hôte, qui est un sur ensemble du jeu de caractères de base, cette tâche ayant été plutôt confiée aux caractères larges qui sont donc tenus, si vous avez bien suivi, de pouvoir représenter n'importe quel caractère multioctet (... à leur manière !), alors qu'on recourrait éventuellement à un tableau si l'on utilisait le type char.
Il existe plusieurs fonctions dédiées à la manipulation des caractères multioctets, déclarées dans stdlib.h. Leur comportement dépend bien évidemment de la valeur du jeu de paramètres LC_CTYPE. En particulier, la fonction **mbstowcs** (**m**ulti**b**yte **s**tring **to** **w**ide **c**har **s**tring) :

```
size_t mbstowcs(wchar_t * pwcs, const char * s, size_t nwcs);
```

permet de convertir une chaîne de caractères multioctets en une chaîne équivalente de caractères larges. La fonction **wcstombs** réalise l'inverse. Par exemple :

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

#define COUNT(t) (sizeof(t) / sizeof(t[0]))

int main()
{
    char s[] = "Hello, world!";
    wchar_t wcs[20];

    mbstowcs(wcs, s, COUNT(wcs));
    wprintf(L"%s\n", wcs);

    return 0;
}
```

VI-E - Recherche dans une chaîne

La fonction strchr

```
char * strchr(const char * s, int c);
```

retourne l'adresse de la première occurrence du caractère c dans la chaîne s ou NULL si aucune occurrence n'a été trouvée. La fonction strrchr (r pour reverse) fait aussi la même chose sauf qu'elle effectue la recherche dans le sens inverse (c'est-à-dire à partir de la fin).

La fonction strpbrk

```
char * strpbrk(const char * s, const char * cs);
```

retourne l'adresse de la première occurrence de n'importe caractère de la chaîne cs dans la chaîne s ou NULL si aucune occurrence n'a été trouvée.

La fonction strstr

```
char * strstr(const char * s, const char * substr);
```

retourne l'adresse de la première occurrence de la chaîne substr dans la chaîne s ou NULL si aucune occurrence n'a été trouvée.

La fonction strtok

```
char * strtok(const char * s, const char * separators);
```

utilisée en boucle, permet d'obtenir toutes les sous chaînes (tokens) de la chaîne s délimitées par n'importe quel des caractères de la chaîne separators. Si s n'est pas NULL, elle est copiée dans une variable locale statique à la fonction. Celle-ci retournera ensuite la première sous chaîne trouvée (en la terminant par '\0'). Si s est NULL et que la fonction a déjà été correctement initialisée, la sous chaîne suivante est retournée, sinon, le comportement est indéfini. Lorsque aucune sous chaîne n'a pu être trouvée, NULL est retournée. Bien entendu, on peut aussi changer la liste des séparateurs à chaque appel de la fonction. Par exemple :

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

int main()
{
    char lpBuffer[100], separators[] = " \t.,!?\n", *p;

    printf("Entrez une chaine de caracteres : ");
    fgets(lpBuffer, sizeof(lpBuffer), stdin);

    printf("\n");

    p = strtok(lpBuffer, separators);
    while(p != NULL)
    {
        printf("%s\n", p);
        p = strtok(NULL, separators);
    }

    return 0;
}
```

VI-F - Conversion des chaînes

Ces fonctions permettent de constituer une valeur numérique (entier ou flottant) à partir d'une chaîne de caractères. Elles sont déclarées dans le fichier **stdlib.h**. On distingue deux grandes familles de fonctions de conversion à savoir les fonctions ato*xx* et celle des fonctions strto*xx*.

```
int atoi(const char * s);
long atol(const char * s);
double atof(const char * s);
```

Par exemple :

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

int main()
{
    int mille = atoi("1000");

    printf("mille = %d\n", mille);
```

```
    return 0;
}
```

Mais que ce passerait-il si on passait par exemple "vive le vent" au lieu de "1000" à la fonction ? Et bien, c'est justement là le problème avec les fonctions ato*xx* : elles ne sont pas capables de signaler qu'on leur a passé n'importe quoi. Dans un programme solide, il faut utiliser les fonctions strto*xx* puis tester si la conversion s'est bien déroulée. Une conversion s'est bien déroulée si elle s'est arrêtée à la rencontre du caractère de fin de chaîne.

```
long strtol(const char * s, char * *p_stop, int base);
unsigned long strtoul(const char * s, char * *p_stop, int base);
double strtod(const char * s, char * *p_stop);
```

Un petit exemple :

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

int main()
{
    char s[] = "FF";
    char * p_stop;
    int n, base = 16;

    n = (int)strtol(s, &p_stop, base);

    if (*p_stop == '\0') /* OK */
        printf("%s (base %d) = %d (base 10).\n", s, base, n);
    else
        printf("Erreur : %c n'est pas un chiffre.\n", *p_stop);

    return 0;
}
```

Pour créer des applications robustes, on n'effectue jamais de saisie à l'aide de scanf (et moins encore à l'aide de gets). On saisit tout d'abord une chaîne (à l'aide de fgets) puis éventuellement la convertir dans le type qu'on veut avoir.

VI-G - Fonctions de manipulation de chaîne « généralisées »

L'intérêt des fonctions **mem*xxx*** (memcpy, memcmp, etc.) est de permettre d'utiliser n'importe quel pointeur sur un objet dans une fonction assez voisine des fonctions orientées chaînes telles que strcpy, strcmp, etc. Elles sont également déclarées dans le fichier **string.h**.

```
void * memcpy(void * p_dest, const void * p_src, size_t n_bytes_to_be_copied);
void * memmove(void * p_dest, const void * p_src, size_t n_bytes_to_be_moved);
void * memcmp(void * p_dest, const void * p_src, size_t n_bytes_to_be_compared);
void * memchr(const void * p_buffer, int c, size_t n_buffer_length);
void * memset(void * p_buffer, int c, size_t n_bytes_to_be_set);
```

La différence entre memcpy et memmove vient du fait que cette dernière peut être utilisée même si les deux buffers (src et dest) se chevauchent. Elle est généralement utilisée pour « glisser » (ou « déplacer ») des données d'où son nom. Par exemple, le programme suivant copie chaque octet du tableau t2 vers le tableau t1 (t1 doit donc être au moins aussi grand que t2).

```
#include <stdio.h>
#include <string.h>
int main()
{
    int t1[10], t2[] = {0, 10, 20, 30, 40};
    int i;

    memcpy(t1, t2, sizeof(t2));
```

```
    for(i = 0; i < 5; i++)
        printf("t1[%d] = %d\n", i, t1[i]);

    return 0;
}
```

VI-H - La date et l'heure

VI-H-1 - Généralités

Les fonctions, macros et types de données standard permettant de travailler avec la date et l'heure sont déclarées/définies dans le fichier **time.h**.

VI-H-2 - Connaître la date et l'heure courantes

La fonction permettant de connaître la date et l'heure calendaires (un "**timestamp**"), est **time** :

```
time_t time(time_t * p_time);
```

En cas d'erreur, (time_t)(-1) est retourné.
On peut donc récupérer la date et l'heure soit en stockant la valeur de retour de time dans une variable de type time_t, soit en passant en argument l'adresse d'une variable du même type. On peut même encore compliquer ...
Il est également important d'avoir à l'esprit que ce n'est qu'une simple et petite fonction, pas un sorcier ni un devin, et qu'elle ne peut donc retourner que ce que le système lui a fournit et rien de plus.
D'autre part, une valeur de type time_t c'est quand même difficile à décoder (en fait pas si difficile que ça mais bref ...) pour un humain. L'idéal pour nous c'est d'avoir une structure avec des champs comme heure, minute, seconde, etc. Heureusement pour nous, cette structure existe. Il s'agit de la structure **struct tm**. Oui, mais comme vous le savez bien, il y a d'un côté le temps local (en anglais **Local Time Zone** ou tout simplement local time) qui dépend de la localisation et de l'autre le Temps Universel (ou **UTC** pour **Coordinated Universal Time**). La structure struct tm est définie dans time.h comme suit :

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour; /* 0 - 23 */
    int tm_mday; /* 1 - 31 */
    int tm_mon; /* 0 - 11 (0 = janvier)*/
    int tm_year; /* annees ecoulees depuis 1900 */
    int tm_wday; /* 0 - 6 (0 = dimanche) */
    int tm_yday; /* 0 - 365 (0 = 1er janvier) */
    int tm_isdst; /* DST (Daylight-Saving Time) flag */
};
```

Le DST flag indique si les informations contenues dans la structure tiennent compte de l'heure d'été ou non. Sa valeur est 1 pour oui, 0 pour non et un nombre négatif si l'information n'est pas disponible.
La fonction **localtime** permet de convertir le temps calendaire (time_t) en temps local. La fonction **gmtime** (de Greenwich Meridian Time, l'ancienne appellation du Temps Universel) permet quant à elle de convertir le temps calendaire en Temps universel. Ces deux fonctions ont le même prototype.

```
struct tm * localtime(time_t * p_time);
struct tm * gmtime(time_t * p_time);
```

En cas d'erreur, NULL est retourné.
Voici un exemple de programme qui affiche la date et l'heure courantes :

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

```
int main()
{
    time_t t;
    struct tm * ti;
    const char * days[] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"};
    const char *
 months[] = {"janvier", "fevrier", "mars", "avril", "mai", "juin", "juillet", "aout", "septembre", "octobre", "no

    t = time(NULL);
    ti = localtime(&t);
    printf("On est le %s %d %s %d\n", days[ti->tm_wday], ti->tm_mday, months[ti->tm_mon],
 ti->tm_year + 1900);
    printf("Et il est %02d:%02d:%02d\n", ti->tm_hour, ti->tm_min, ti->tm_sec);

    return 0;
}
```

On a également la fonction **asctime** qui permet de convertir facilement un temps représenté par une structure (struct tm) en une chaîne de caractères et la fonction **ctime** qui reçoit quant à elle un argument de type time_t. Dommage que le format utilisé soit anglo-saxon.

```
char * asctime(const struct tm * pTime);
char * ctime(const time_t * p_time);
```

Par exemple :

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

int main()
{
    time_t t = time(NULL);

    printf("Date is : %s\n", ctime(&t));

    return 0;
}
```

Il y a également la fonction **strftime** assez complexe pour être détaillée ici.

VI-H-3 - Les conversions date <-> timestamp

Nous avons vu tout à l'heure les fonctions localtime et gmtime qui permettent de connaître les informations de date et heure à partir d'un temps calendaire. Il existe une fonction **mktime** qui réalise l'inverse, c'est-à-dire qui calcule le temps calendaire en fonction d'informations de date et heure **locales**. Lors d'un appel à mktime, les champs tm_wday et tm_yday sont ignorés. En effet, ils peuvent être calculés à partir des valeurs des autres champs.
Voici un exemple de programme qui demande à l'utilisateur d'entrer une date et qui affiche le jour de la semaine correspondant à cette date.

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

int main()
{
    struct tm ti;

    printf("Entrez une date au format jj/mm/aaaa : ");
    memset(&ti, 0, sizeof(ti));
    if (scanf("%d/%d/%d", &ti.tm_mday, &ti.tm_mon, &ti.tm_year) != 3)
        printf("Mauvais format.\n");
    else
    {
        time_t t;
```

```
        const char *
 days_tab[] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"};

        ti.tm_mon -= 1;
        ti.tm_year -= 1900;

        t = mktime(&ti);
        memcpy(&ti, localtime(&t), sizeof(struct tm));

        printf("C'est un %s.\n", days_tab[ti.tm_wday]);
    }

    return 0;
}
```

VI-H-4 - Connaître le temps processeur d'un programme

Une autre fonction bien pratique de time.h est la fonction **clock** qui retourne le temps processeur utilisé par le programme depuis le début de son exécution (le point marquant le "début de l'exécution" d'un programme est défini par l'implémentation et non par le C), exprimé en **tops**, un top étant égal à 1/CLOCKS_PER_SEC seconde (il s'agit là d'une division de réels). Le prototype de cette fonction est :

```
clock_t clock(void);
```

clock_t étant évidemment un type arithmétique représentant le temps d'utilisation du processeur d'un programme depuis le début de son exécution. Une des multiples applications de cette fonction (mais qui n'est pas la meilleure ...) est de ne quitter une boucle (qui ne fait rien de spécial ...) que lorsqu'un certain temps s'est écoulé, cela permet de suspendre le programme pendant un intervalle de temps donné. Par exemple :

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

void wait(int ms);

int main()
{
    printf("Suspension du programme pendant 5s.\n");
    wait(5000);
    printf("Termine.\n");

    return 0;
}

void wait(int ms)
{
    clock_t start = clock();
    while ((clock() - start) * 1000 < ms * CLOCKS_PER_SEC) ;
}
```

VI-I - Les nombres pseudo-aléatoires

La bibliothèque standard du langage C est fournie avec des fonctions permettant de générer des nombres de façon pseudo aléatoire. Le générateur de ces nombres doit tout d'abord être initialisée avec une graine qui doit elle-même être plus ou moins aléatoire. Si la graine est constante, la même série de nombres va être générée chaque fois que le programme sera exécuté car le générateur sera alors toujours initialisé avec la même valeur. En ce qui nous concerne, pas la peine d'aller chercher sur Mars, le temps est une valeur qui change « tout le temps » ! Elle peut donc convenir dans la plupart des cas pour initialiser le générateur.
Les fonctions liées aux nombres aléatoires, déclarées dans stdlib.h sont :

```
void srand(unsigned int seed);
int rand(void);
```

La fonction **srand** permet d'initialiser le générateur. La fonction **rand** retourne à chaque appel un nombre pseudo aléatoire compris entre 0 et **RAND_MAX** (qui doit être au moins égal à 32767 quel que soit l'implémentation). Par exemple :

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

int main()
{
    int i;

    srand((unsigned)time(NULL));

    printf("Voici dix nombres tires au hasard compris entre 0 et 99.\n");

    for(i = 0; i < 10; i++)
        printf("%d\t", rand() % 100);

    return 0;
}
```

VII - Pour conclure

Maîtriser les concepts avancés du langage C n'est pas du tout difficile une fois qu'on ait bien compris les bases du langage. Comme il a été dit depuis le départ, ce tutoriel n'est pas complet. Vous pouvez cependant vous assurez que vous savez écrire maintenant écrire du correct en C, et en savez également assez pour développer des applications bien structurées et robustes à l'aide de ce langage.