Les types et les variables du langag C

Le préprocesseur C
Les expressions et les opérateur

Accès rapide :

Nous allons, tout au long de ce chapitre, étudier l'ensemble des types de données que fournit le langage C ainsi que l'utilisation de variables basées sur ces types.

La notion de type de données

Un type de données représente un ensemble de valeurs d'une même nature. Pour faire simple, on peut parler du type entier (qui représente l'ensemble des valeurs numériques entières : par exemple : 1, 2, 3, 4, ...), du type flottant (3.1415926, 4.4, ...), des booléens (état vrai/faux), ... Le langage C propose un certain nombre de types prédéfinis. Il est même possible de définir vos propres types de données (par le biais, notamment, de la notion de structure sur laquelle nous reviendrons ultérieurement).

En C la notion de type est portée par les variables et non par les valeurs (contrairement à des langages comme Javascript ou Python). Autrement dit, à la seule connaissance d'une valeur, on sera incapable d'en extraire son type. Par contre le compilateur saura déterminer si une valeur est correctement utilisée, pas le biais de la variable qui la contiendra. Une variable peut donc être vue comme un conteneur qui ne peut recevoir que des valeurs d'un type bien défini : le type d'une variable, comme nous allons le voir, est fixé dès la déclaration de celle-ci.

Remarque : il est important de noter que, selon les différentes plates-formes, le nombre d'octets utilisé pour stocker une valeur associée à un type de donné peut varier. Cette particularité peut poser un problème en matière de portabilité d'un programme. Pour connaitre le nombre d'octets utilisé par une variable (ou un type de données), vous pouvez utiliser l'opérateur sizeof (la valeur retournée sera bien un nombre d'octets, et non pas un nombre de bits).

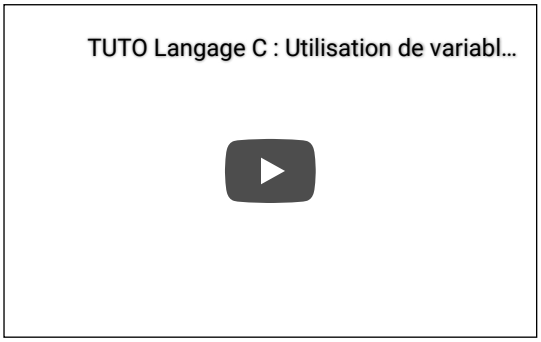
```
#include <stdio.h>
#include <stdlib.h>
```

```
    printf( "sizeof(long) == %d bytes\n",

    return 0;
}
```

Sur une architecture PC 64 bits, voici les résultats produits, mais notez bien que si vous changez d'architecture (16 bits ou 32 bits) certaines valeurs pourront changer.

```
sizeof(char) == 1 bytes
sizeof(short) == 2 bytes
sizeof(int) == 4 bytes
sizeof(long) == 8 bytes
```



Déclaration de variables

Pour pouvoir utiliser une variable en C, il faut, en premier lieu, la déclarer : c'est-à-dire lui donner un nom et un type. Nous reviendrons par la suite sur l'ensemble des types prédéfinis que peut prendre une variable. Mais dans un premier temps, nous allons voir que nommer une variable doit répondre à quelques règles bien précises.

Correctement nommer ses variables

Un nom de variable peut être constitué de lettres, de chiffres et du caractère blanc souligné (underscore en anglais). Il est donc inutile d'y placer d'autres caractères car votre compilateur produira inexorablement une erreur. De plus, un nom de variable ne peut commencer que soit par une lettre, soit par un blanc souligné (mais en aucun cas un chiffre).

Remarque : en C, il y a une différence entre majuscules et minuscules. Les deux noms de variables `toto` et `Toto` identifieront donc bien deux variables distinctes.

Vous pouvez donner n'importe quel nom à une variable pourvu qu'il respecte les règles précédentes et à condition qu'il n'existe aucun mot clé du langage qui porte ce nom (aux majuscules et minuscules près). Néanmoins, et dans un souci de lisibilité, il est très fortement conseillé de choisir des noms de variables clairs et explicites (en rapport avec l'utilisation que vous souhaitez en faire). Appeler ses variables `v1`, `v2`, `v3`, ... n'est donc pas une très bonne idée.

Remarque : plus le temps passe, plus les programmes deviennent volumineux. Le plus gros chalenge reste donc de comprendre et de maintenir ces codes (d'autant que ces codes peuvent avoir été écrits pas d'autres personnes) le plus rapidement possible. Le choix de vos noms de variables et tant qu'on y est, le choix des noms de vos fonctions, représentent une activité très importante.

La syntaxe d'une déclaration

Deux possibilités vous sont offertes pour déclarer une variable. Soit vous déclarer votre variable sans l'initialiser et dans ce cas vous ne pouvez pas présager de sa valeur (la zone mémoire est juste

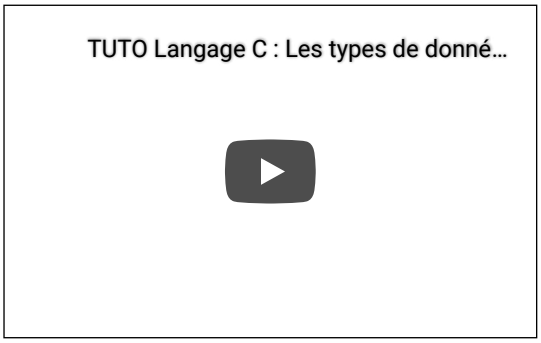
Dans les deux cas, il vous faudra aussi typer la variable et terminer votre déclaration par un caractère « ; ». Les deux formes de syntaxe générale vous sont proposées ci-dessous.

```
type variableName;
type variableName = initialValue;
```

Voici trois exemples concrets de déclarations de variable. La première ligne déclare une variable entière nommée `count` : elle est initialisée à la valeur 5. La seconde ligne déclare une valeur, nommée `price`, de type flottant à double précision : cette variable est initialisée à la valeur 10.99. Enfin la dernière ligne de code déclare un variable `character` non initialisée (ne présagez surtout pas du fait que çà valeur initial est 0, est çà risques de ne pas être le cas).

```
int count = 5;
double price = 10.99;
char character;
```

Les types



numériques

Bien entendu, le langage permet la manipulation de nombres numériques. Mais il existe plusieurs types de nombres. Ils sont séparés en deux catégories : les nombres entiers et les nombres décimaux (avec des chiffres derrière la virgule).

Les types numériques entiers

Cet ensemble de types est lui-même divisé en deux catégories de types : les types entiers signés et les type entiers non signés. Dans les deux cas, les noms des types existants sont les mêmes (`char`, `short`, `int` et `long`) : la seule différence réside dans le fait que les types numériques entiers non signés sont préfixés du mot clé `unsigned`.

Là où les choses se compliquent, c'est que le nombre d'octets utilisé par une donnée d'un type particulier est dépendant de la plate-forme d'exécution. On ne peut donc pas simplement dire qu'une donnée de type `int` va prendre x octets : il faut préciser la nature de la plate-forme. Ainsi sur une plate-forme de type PC et basées sur une architecture 64 bits, le type `int` prendra quatre octets (rappel : pour connaître la taille d'un type, utilisez l'opérateur `sizeof`). La seule exception est pour les types `char` et `unsigned char` qui, eux, prendront toujours un octet.

En conséquence voici ce que garantie la norme sur le langage C. Les types `char` ou `unsigned char` occuperont un octet. Les types `short` et `unsigned short` auront une taille supérieure ou égale à celle de `char` ou `unsigned char`. Les types `int` et `unsigned int` auront une taille supérieure ou égale à celle de `short` ou `unsigned short`. Et enfin, les types `long` et `unsigned long` auront une taille supérieure ou égale à celle de `int` ou `unsigned int`.

Remarque : sur-typer vos variables peut engendrer une surconsommation mémoire importante. Il est donc important de

celui de la variable de destination ne posera aucun problème et compilera sans besoin de syntaxe complémentaire.

```
int littleVariable = 100;
long bigVariable;

bigVariable = littleVariable; /* Conversio
```

Il en va de même avec cette seconde variante :

```
int littleVariable = 100;
long bigVariable = littleVariable; /* Conv
```

Par contre, dans l'autre sens, les choses seront un petit peu plus compliquées. Effectivement stocker, par exemple, une donnée occupant 4 octets dans un nouvel espace mémoire de deux octets peut être dangereux. Dans de tels cas, des warnings seront déclenchées car les données risqueront d'avoir étaient tronquées. Il n'est pas bon de laisser un programme compiler avec des warnings. Une bonne pratique de développement est de garantir que votre programme se compilera en provoquant zéro erreur et zéro warning.

Si vous êtes certains que le changement de type n'engendrera pas de problème, il vous est alors possible de réaliser un « cast » (en français, on préférera le mot « transtypage »). Cela se fait en utilisant l'opérateur `(char)`.

```
int bigVariable = 100;
char littleVariable = bigVariable;
char littleVariable2 = (char) bigVariable;
```


Vous pouvez exprimer des nombres entiers dans trois bases numériques différentes : en octal (le nombre est simplement préfixé d'un caractère 0), en décimal (certainement l'utilisation la plus classique), mais aussi en hexadécimal (préfixé des deux caractères 0x). A titre d'exemple, voici un petit programme montrant l'utilisation de ces possibilités.

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

int main() {

    printf( "Un entier exprimé en décimal
    printf( "Un entier exprimé en octal :
    /* printf( "Un entier exprimé en octal
    printf( "Un entier exprimé en octal :
    printf( "Un entier exprimé en hexadéci

    return EXIT_SUCCESS;
}
```

Les résultats produits par cet exemple sont les suivants :

```
$> gcc -o Sample Sample.c
$> Sample
Un entier exprimé en décimal : 33
Un entier exprimé en octal : 7
Un entier exprimé en octal : 8
Un entier exprimé en hexadécimal : 255
$>
```

Enfin, pour clore cette section sur les nombre entiers, notez qu'une constante entière est forcément typée `int`. Ce que je veux sous-entendre, c'est que la définition suivante ne compilera pas : `long value = 10000000000` (on part du principe qu'on est sur une plate-forme 64 bits). Effectivement un `int` y occupera normalement 4

(L pour long, bien entendu) :

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

int main() {

    long value = 10000000000L;
    printf( "Un long, initialisé à partir

    return EXIT_SUCCESS;
}
```

Les types numériques décimaux

Le langage C met à votre disposition deux types numériques décimaux (aussi appelés types numériques flottants). Le type `float` pour les nombres décimaux en simple précision et `double` pour les nombres décimaux en double précision. Encore une fois le nombre d'octets utilisés pour représenter des données basées sur ces types est dépendant de la plate-forme utilisée. Sur des plates-formes type PC (architectures 32 bits ou 64 bits), ils occupent 4 octets (`float`) et 8 octets (`double`).

Vous n'avez pas à utiliser les mots clés `signed` ou `unsigned` avec les nombres flottants de C : ils sont obligatoirement signés. Voici quelques exemples de déclaration de valeurs numériques décimales.

```
float f = 3.0;
float constanteFlottante = 5.4321f;
double notationDecimale = 3.14159254;
double notationExponentielle = 3.2e-10;
```


Remarque : par défaut, lorsque vous saisissez en dur une constante numérique décimale dans un code C, cette constante est typée `double`. Si vous souhaitez définir une constante de type `float`, vous pouvez utiliser la syntaxe proposée sur la seconde ligne de l'exemple ci-dessus (`5.4321f`). Dans certaines situations, cette syntaxe vous sera utile.

Il possible de transformer des valeurs entières en valeurs flottantes : dans ce cas, les choses peuvent être définie implicitement. Il est aussi possible de transformer des valeurs flottantes en valeurs entières, mais dans ce cas, une perte de précision pourra être constatée : par exemple transformer la valeur PI en une valeur entière nous fera perdre la partie décimale. Il vous faudra alors acter le changement de type de manière explicite en ajoutant un opérateur de cast (de transtypage, en français). Voici quelques exemples :

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

int main() {

    float initialFloat = 3.0f;
    double initialDouble = 3.14159254;
    int initialInteger = 345;

    float firstTransformation = initialInt
    double secondTransformation = initialI
    int thirdTransformation = (int) initia
    int forthTransformation = (int) initia

    printf( "%f - %lf - %d - %d\n",
        firstTransformation, secondTransfo
        thirdTransformation, forthTransfor

    return EXIT_SUCCESS;
}
```

Néanmoins, le standard C dit que toute valeur entière différente de 0 sera considérée comme vrai (en cas de test logique) et que 0 sera considéré comme faux. L'exemple suivant fonctionne donc parfaitement, en considérant la règle énoncée ci-dessus.

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


int main() {

    /* Exemple de définition d'un pseudo b
    int state = 1;    /* ie true */

    if ( state ) {
        printf( "true\n" );
    } else {
        printf( "false\n" );
    }

    /* Exemple de boucle finissant avec la
    int a = 3;
    while( a ) {
        printf( "a == %d\n", a );
        a--;
    }

    return EXIT_SUCCESS;
}
```

Fort de ce constat, il est malgré tout possible d'améliorer considérablement la compréhension de votre code. Effectivement, depuis C11 (standard 2011 de C), une nouvelle entête vient de faire son apparition : `<stdbool.h>`. Elle définit un ensemble de macros portant des noms bien plus explicite. Ces macros sont au nombre de trois : `bool` (un alias sur int), `true` (définit sur la valeur 1) et `false` (définit sur la valeur 0). Vous trouverez ci-dessous le même code mais légèrement remanié afin d'en améliorer sa lisibilité.


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

int main() {

    /* Exemple de définition d'un pseudo b
    bool state = true;

    if ( state ) {
        printf( "true\n" );
    } else {
        printf( "false\n" );
    }

    /* Exemple de boucle finissant avec la
    int a = 3;
    while( a ) {
        printf( "a == %d\n", a );
        a--;
    }

    return EXIT_SUCCESS;
}
```

Attention, pour compiler ce programme, vous êtes obligé de demander à votre compilateur d'élever son niveau de compatibilité sur C11. Pour ce faire, ajoutez l'option `-std=c11` sur la ligne de commande de démarrage de vote compilateur.

```
$> gcc -o Sample -Wall -std=c11 Sample.c
$> ./Sample
true
a == 3
a == 2
```

Utilisation de caractères en C

Tout comme pour les booléens, le type permettant de manipuler les caractères est un peu ambiguë. Effectivement, il n'y a pas de différence entre le type entier numérique sur un octet (`char`) et les caractères. Ainsi quand vous aurez une variable typée `char`, il sera de votre responsabilité de savoir s'il faut l'interpréter en tant qu caractère ou en tant qu'octet. L'exemple ci-dessous joue avec cett ambiguïté : notez la syntaxe utilisé pour spécifier la valeur numérique d'un caractère ('a' par exemple).

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

int main() {

    char numericCode1 = 65;
    char numericCode2 = 'a';

    printf( "numericCode1 as byte : %d\n",
    printf( "numericCode1 as char : %c\n",

    printf( "numericCode2 as byte : %d\n",
    printf( "numericCode2 as char : %c\n",

    return EXIT_SUCCESS;
}
```

Et voilà ce que donne l'exécution de ce programme.

```
$> gcc -o Sample -Wall Sample.c
$> ./Sample
numericCode1 as byte : 65
numericCode1 as char : A
numericCode2 as byte : 97
numericCode2 as char : a
$>
```


Utilisation de tableaux en C

Comme beaucoup d'autres langages de programmation, le langage C défini la notion de tableaux de données. Un tableau de données permet de contenir plusieurs sous-éléments d'un même type. Il est donc nécessaire de typer vos tableaux : c'est à dire définir la nature des éléments qu'il va contenir. L'exemple ci-dessous défini un tableau de 5 entiers, initialise chaque élément du tableau puis finit pas afficher chaque valeur stockées dans le tableau.

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

#define ARRAY_SIZE 5

int main() {

    int array[ ARRAY_SIZE ];

    array[0] = 1;
    array[1] = 2;
    array[2] = 4;
    array[3] = 8;
    array[4] = 16;

    for( int i=0; i<ARRAY_SIZE; i++ ) {
        printf( "array[%d] == %d\n", i, ar
    }

    return EXIT_SUCCESS;
}
```

(type int) et aura comme taille 5 (cette valeur étant déclarée dans la macro ARRAY_SIZE). La taille du tableau doit toujours être spécifiée entre crochets.

Attention, les indices d'un tableau (les positions des éléments qu'il contient) sont obligatoirement exprimés à partir de 0. Cet indice 0 correspond donc à la position du premier élément. En conséquence, les indice vont donc de 0 à n-1 (ou n correspond la taille du tableau). Les cinq lignes de code qui suivent donne donc une valeur à chaque case du tableau (pour information, nous stockons dans ce tableau les premières puissances de 2). Comme pour la ligne précédente, les crochets sont utilisés pour déterminer à quelle position travailler.

Enfin, nous utilisons une boucle for pour parcourir toutes les positions du tableau et en récupérer les valeurs. Attention, la boucle proposée utilise un détail de syntaxe qui a été rajouté dans la version 2011 du langage C. Pour correctement compiler cet exemple, il est nécessaire d'utiliser la ligne de commande suivante :

```
$> gcc -o Sample -Wall -std=c99 Sample.c
$> ./Sample
array[0] == 1
array[1] == 2
array[2] == 4
array[3] == 8
array[4] == 16
$>
```

Attention : trois remarques sur l'utilisation de tableaux de données en C sont à mémoriser par coeur :

- Un tableau C ne mémorise pas le nombre d'éléments qu'il contient. Il est donc de votre responsabilité de vous en souvenir. Dans l'exemple précédent, cela est réalisé via la déclaration de la macro ARRAY_SIZE qui est fixée à cinq. Il est donc possible dans la boucle for de lui repasser cette valeur.
- Il est de votre responsabilité de garantir que vos indices dans le tableau soient cohérents. Si vous cherchez à accéder à une zone mémoire en dehors du tableau, votre programme se contentera d'utiliser cette zone mémoire, sauf que très certainement une donnée non souhaitée qui se trouve à cet endroit. Vous ne contrôlerez donc pas la valeur retournée pour cet espace mémoire. Pire encore, si vous cherchez à modifier cette zone mémoire, vous modifierez alors certainement le contenu d'une autre variable (laquelle, ????).
- L'initialisation des cases du tableau n'est pas faites par défaut. Si vous oubliez de donner un état initial à chacune des cases de votre tableau, vous ne pourrez plus présager des valeurs qui y seront stockées. Effectivement dans notre exemple, le tableau est physiquement stocké dans ce qu'on appelle la pile d'exécution de votre programme : cette zone mémoire peut avoir été préalablement manipulée et vous ne pouvez pas systématiquement garantir l'état de cette zone. Si nous supprimons les lignes d'initialisation, alors vous ne saurez pas dans quel état seront vos entiers. Les plus audacieux d'entre vous aurons, à coup sûr, testé la chose. Si d'aventure cinq valeurs 0 vous étaient retournées alors soit vous avez beaucoup de chance, soit vous compilez en mode debug (repasser alors en mode release pour constater le problème).

Bien entendu, vous auriez aussi pu initialiser chacune des cases du tableau via une boucle for. L'exemple précédent est donc équivalent à celui qui suit.

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

```
    for( int i=0; i<ARRAY_SIZE; i++ ) {
        array[i] = pow( 2, i );
    }

    for( int i=0; i<ARRAY_SIZE; i++ ) {
        printf( "array[%d] == %d\n", i, ar
    }

    return EXIT_SUCCESS;
}
```

Attention, la librairie mathématique n'est pas liée par défaut à votre exécutable. Il est donc nécessaire de lancer la ligne de compilation suivante (c'est l'option `-lm` qui demande à lier la librairie mathématique) :

```
$> gcc -o Sample -Wall -std=c99 Sample.c -
$> ./Sample
array[0] == 1
array[1] == 2
array[2] == 4
array[3] == 8
array[4] == 16
$>
```

Il est aussi possible de définir l'état initial des cases d'un tableau directement à sa déclaration. Dans ce cas, il va nous falloir utiliser une nouvelle syntaxe à base d'accolades. Dans un tel cas, il n'est plus nécessaire de spécifier explicitement la taille du tableau, car elle peut être déduite du nombre de valeurs initiales spécifiées. Bien entendu le type des valeurs initiales doit être compatible avec le type de déclaration du tableau. Voici encore une fois un exemple de code équivalent.


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

#define ARRAY_SIZE 5

int main() {

    int array[] = { 1, 2, 4, 8, 16 };

    for( int i=0; i<ARRAY_SIZE; i++ ) {
        printf( "array[%d] == %d\n", i, ar
    }

    return EXIT_SUCCESS;
}
```

Dernière remarque quant à l'utilisation de tableau en C : les tableaux multidimensionnels ne sont pas supportés par le langage C. Il est néanmoins possible de définir des tableaux de tableaux. L'exemple ci-dessous définit une matrice 8x8 : cela peut notamment servir de base pour le codage d'un jeu d'échec.

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

#define BOARD_WIDTH 8
#define BOARD_HEIGHT 8

int main() {

    int chessBoard[8][8] = {
        { 2, 3, 4, 6, 5, 4, 3, 2 },
        { 1, 1, 1, 1, 1, 1, 1, 1 },
        { 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0 },
        { 1, 1, 1, 1, 1, 1, 1, 1 },
```

```
            printf( "%d ", chessBoard[y][x
        }
        puts( "" );
    }

    return EXIT_SUCCESS;
}
```

Et voici le résultat produit par cet exemple :

```
$> gcc -o Sample -Wall -std=c99 Sample.c
$> ./Sample
2 3 4 6 5 4 3 2
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
$>
```

Utilisation de chaînes de caractères

Il n'y a pas de type chaîne de caractères a proprement parlé en C. Une chaîne sera simplement un tableau de caractères au point près qu'un caractère ASCII de code 0 doit forcément terminer cette chaîne. Comme nous le verrons dans la section suivante, tableaux et pointeurs sont deux facettes d'un même concept (du moins en C). L'usage C veut que ce soit la syntaxe pointeur qui soit utilisée la plupart du temps (mais cela reste un tableau de caractères). Voici un exemple qui déclare deux variables de type chaîne de caractères et les affiche sur la console.


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

int main() {

    const char * firstName = "Diego";
    const char * lastName = "De La Vega";

    printf( "Zorro is ... %s %s\n", firstN

    return EXIT_SUCCESS;
}
```

Remarque : notez que les deux valeurs d'initialisation de nos variables sont placées entre des doubles guillemets.

Seconde remarque : le qualificateur `const` est rajouté devant le type de la variable pour indiquer que cette variable est non modifiable. Si vous cherchiez à modifier la chaîne, une erreur de compilation serait alors produite.

Voici le résultat produit par cet exemple :

```
$> gcc -o Sample -Wall Sample.c
$> ./Sample
Zorro is ... Diego De La Vega
$>
```

Voici un second exemple de code qui, cette fois ci, cherche à produire une chaîne caractère par caractère (elle ne peut donc pas être constante). Il convient donc de ne pas oublier de clore cette chaîne avec un caractère de code ASCII nul (caractère '\0'). Histoire de mixer les styles de codage, nous utiliserons plutôt la syntaxe tableau (à la place de la syntaxe pointeur). Pour autant, le programme fini par afficher cette chaîne via la fonction `printf` en utilisant le format `%s` (pour string == chaîne de caractères).

```
    char buffer[12];    /* 12 caractères *
    buffer[0]  = 'H';
    buffer[1]  = 'e';
    buffer[2]  = 'l';
    buffer[3]  = 'l';
    buffer[4]  = 'o';
    buffer[5]  = ' ';
    buffer[6]  = 'W';
    buffer[7]  = 'o';
    buffer[8]  = 'r';
    buffer[9]  = 'l';
    buffer[10] = 'd';
    buffer[11] = '\0';

    printf( "%s\n", buffer );

    return EXIT_SUCCESS;
}
```

Voici le résultat produit par ce nouvel exemple :

```
$> gcc -o Sample -Wall Sample.c
$> ./Sample
Hello World
$>
```

Enfin, une librairie faisant partie du standard C permet la manipulation de vos chaînes de caractères en proposant un certain nombre de fonctions prédéfinies. Cette libriaire se nomme `<sdtio.h>`. Nous verrons comment utiliser ces fonctions au fur et à mesure des chapitres de ce tutorial.

La notion de pointeurs

Le simple fait d'énoncer ce concept que déjà certains d'entre vous paniquent. Il ne faut pas : on se fait souvent tout un monde d'une chose qui n'est pas si compliquée que cela. En fait, un pointeur est une adresse en mémoire : c'est tout. Par contre, il est vrai que l'on peut faire de nombreuses choses plus ou moins subtiles si l'on manipule des adresses mémoire. L'exemple suivant commence par calculer l'adresse en mémoire d'une variable (via l'opérateur `&` placé devant le nom de la variable) : cette adresse est stockée dans une autre variable nommée `addr` et elle est typée comme étant un pointeur sur un entier (une adresse vers un entier). On peut ainsi considérer que nous venons de faire un alias vers une variable. Si on utilise cet alias, on peut alors modifier le contenu de la variable d'origine. C'est ce que fait la ligne suivante : `*addr` correspondant au contenu de la zone mémoire pointée (donc, un entier dans notre exemple).

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

int main() {

    int value = 33;
    int * addr = &value;
    *addr = *addr + 1;

    printf( "value == %d\n", value );

    return EXIT_SUCCESS;
}
```

Note : on reconnait une variable de type pointeur par le biais du caractère `*` placé entre le type et le nom de la variable.

Voici le résultat produit par ce nouvel exemple :

```
$> gcc -o Sample -Wall Sample.c
$> ./Sample
value == 34
```

premier caractère. Ce premier caractère (comme les autres d'ailleurs) étant de type `char`.

Nous reviendrons ultérieurement, et à de nombreuses reprises, sur ce concept de pointeurs très important en C.

Définition de structures en C

La notion de structure est très importante en C. Pour information, certains autres langages de programmation (le langage Pascal par exemple) ne parleront pas de structures, mais plutôt d'enregistrements (records en anglais) : c'est le même concept. Une structure correspond à un nouveau type de données qui est définit en agrégeant plusieurs éléments, chacun d'eux ayant leur propre type de données.

Par exemple, imaginons que nous ayons besoin de manipuler le concept de nombre rationnel (une fraction). Nous pouvons alors définir un nouveau type que nous appellerons `Rational`. Il sera constitué de deux entiers : le numérateur et le dénominateur.

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

struct Rational {
    int numerator;
    int denominator;
};


int main() {

    struct Rational r;
    r.numerator = 1;
    r.denominator = 3;

    printf( "Our rational: [%d/%d]\n",
            r.numerator, r.denominator );

    return EXIT_SUCCESS;
}
```


Dans cet exemple, nous avons donc définit notre type de données `Rational`. Notez bien la présence du caractère « ; » en fin de définition de la structure : il est obligatoire. Ensuite, le `main`, créé une variable basée sur ce type. Nous devons réutiliser le mot clé `struct` pour cette déclaration de variable : nous aurions bien une solution pour nous en passer, mais nous verrons çà un peu plus loin dans ce chapitre. Cette variable est donc initialisée avec deux affectations étant donné qu'elle contient deux champs (`numerator` et `denominator`). Enfin, la fonction `printf` nous permet d'en afficher son contenu. Voici le résultat produit par ce programme.

```
$> ./Sample
Our rational: [1/3]
$>
```

Définition de types énumérés en C

Un type énuméré définit un certain nombre d'états autorisés. Normalement, une variable basée sur un tel type ne devrait pas pouvoir prendre un autre état que ceux autorisés par le type énuméré. Malgré cela il en va un peu différemment en C, étant donné qu'un type énuméré est en fait un entier : on va seulement définir un ensemble de constantes associé à certaines valeurs numériques.

Par exemple, imaginons que nous souhaitions définir différents états d'un cycle de vie d'un logiciel. Nous pouvons alors définir un

```
#include <stdlib.h>

enum LifeCycleState {
    INITIALIZED, STARTED, STOPED, DESTROYE
};

int main() {

    enum LifeCycleState currentState = INI
    printf( "Current state : %d\n", curren

    currentState = STARTED;
    printf( "Current state : %d\n", curren

    currentState = STOPED;
    printf( "Current state : %d\n", curren

    currentState = STARTED;
    printf( "Current state : %d\n", curren

    currentState = STOPED;
    printf( "Current state : %d\n", curren

    currentState = DESTROYED;
    printf( "Current state : %d\n", curren

    return EXIT_SUCCESS;
}
```

Voici le résultat produit par le programme ci-dessus : comme dit précédemment, une valeur (un état) définie dans un type énuméré est en fait un entier, d'où le format utilisé dans les appels à la fonction `printf`.

```
$> gcc -Wall -o Sample Sample.c
$> ./Sample
Current state : 0
Current state : 1
Current state : 2
Current state : 1
Current state : 2
Current state : 3
$>
```


Sauf mention de votre part, le premier état de votre type énuméré sera associé à la valeur 0, le suivant à la valeur 1, et ainsi de suite, jusqu'au dernier. Si vous souhaitez changer cela, vous pouvez associer des valeurs numériques différentes à vos états. Si un état est associé à la valeur 10 (par exemple), les états suivants seront associés aux valeurs 11, 12, ... Voici un exemple de spécification de valeurs numériques pour les états de votre type énuméré.

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

enum LifeCycleState {
    INITIALIZED=1, STARTED, STOPED=10, DES
};

int main() {

    enum LifeCycleState currentState = INI
    printf( "Current state : %d\n", curren

    currentState = STARTED;
    printf( "Current state : %d\n", curren

    currentState = STOPED;
    printf( "Current state : %d\n", curren

    currentState = STARTED;
    printf( "Current state : %d\n", curren

    currentState = STOPED;
    printf( "Current state : %d\n", curren

    currentState = DESTROYED;
```

Et voici les résultats qui seront alors produits.

```
$> gcc -Wall -o Sample Sample.c
$> ./Sample
Current state : 1
Current state : 2
Current state : 10
Current state : 2
Current state : 10
Current state : 11
$>
```

Définition d'unions en C

Une union est un type un peu particulier. Elle permet de voir une zone mémoire de différentes manières (via différents types). La taille de cette zone mémoire sera la taille mémoire nécessaire dans le pire des cas considéré (le plus gros des types utilisés dans les branches de l'union). Voici un exemple d'utilisation d'union : il s'inspire de la manipulation des registres d'un CPU.

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

union RegisterType {
    struct {
        unsigned char a, b, c, d;
    } int8;
    struct {
        unsigned short ax, bx;
    } int16;
    struct {
        unsigned int eax;
    } int32;
};


int main() {

    union RegisterType myRegister;
    myRegister.int32.eax = 0xFF00E080;

    printf( "32bits access: %x\n", myRegis
    printf( "16bits access: %x - %x\n", my
    printf( "8bits access: %x - %x - %x -
        myRegister.int8.a, myRegister.int8

    return EXIT_SUCCESS;
}
```

Voici ce que produit l'exemple de code ci-dessus. **Attention** : les résultats produits peuvent être différents en fonction du type de plates-formes utilisé (architecture little indian ou big indian).

```
$> gcc -Wall -o Sample Sample.c
$> ./Sample
32bits access: ff00e080
16bits access: e080 - ff00
8bits access: 80 - e0 - 0 - ff
$>
```

Utilisation du mot clé `typedef`

Le mot clé `typedef` permet de définir des alias de types de données. Il est très utile quand on l'utilise conjointement avec une structure, un type énuméré ou une union. Effectivement, dans les trois cas, vous êtes obligé de réutiliser un mot clé (`struct`, `enum` ou `union`) pour déclarer les variables basées sur ces types. Avec l'emploi du mot clé `typedef`, vous définissez donc un alias, mais

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

struct _Rational {
    int numerator;
    int denominator;
};

typedef struct _Rational Rational;


int main() {

    /* Without typedef usage */
    struct _Rational r1;
    r1.numerator = 1;
    r1.denominator = 3;

    printf( "r1: [%d/%d]\n", r1.numerator,

    /* With typedef usage */
    Rational r2;
    r2.numerator = 4;
    r2.denominator = 5;

    printf( "r2: [%d/%d]\n", r2.numerator,

    return EXIT_SUCCESS;
}
```

Et voici les résultats produits par ce programme.

```
$> gcc -Wall -o Sample Sample.c
$> ./Sample
r1: [1/3]
r2: [4/5]
$>
```


Il est à noter qu'il est possible de combiner la déclaration de la structure et celle de l'alias (introduit par le mot clé `typedef`). C'est très certainement la forme d'utilisation du `typedef` que vous trouverez le plus souvent.

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

typedef struct _Rational {
    int numerator;
    int denominator;
} Rational;


int main() {

    /* Without typedef usage */
    struct _Rational r1;
    r1.numerator = 1;
    r1.denominator = 3;

    printf( "r1: [%d/%d]\n", r1.numerator,

    /* With typedef usage */
    Rational r2;
    r2.numerator = 4;
    r2.denominator = 5;

    printf( "r2: [%d/%d]\n", r2.numerator,

    return EXIT_SUCCESS;
}
```

Utilisation de constantes

basée sur n'importe quel type. Comme une constante ne peut pas changer de valeur d'un appel de fonction à un autre, il est fréquent de trouver les définitions de constantes en variables globales et non en variables locales. Voici quelques exemples de définitions de constantes.

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

/* global constant variable */
const double MY_PI = 3.141592654;


int main() {

    /* local constant variable */
    const unsigned int LOOP_NUMBER = 5;

    for( unsigned int i=0; i<LOOP_NUMBER;
        printf( "%lfx%u == %lf\n", MY_PI,
    }

    return EXIT_SUCCESS;
}
```

Remarque : conventionnellement, les constantes sont souvent orthographiées en majuscules.

Nous compilons maintenant ce programme afin d'en voir les résultats. Néanmoins, attention, cet exemple utilise une boucle `for` proposée dans la version C11 du langage C : il est donc nécessaire de rajouter l'option `-std=c11` sur l'ordre de compilation.

```
$> gcc -o Sample Sample.c -Wall -std=c11
$> ./Sample
3.141593x0 == 0.000000
3.141593x1 == 3.141593
3.141593x2 == 6.283185
3.141593x3 == 9.424778
3.141593x4 == 12.566371
$>
```


Enfin, notez que si vous cherchez à modifier la valeur d'une variable qualifiée de constante, une erreur de compilation sera produite. **Voici un exemple de code ne compilant pas**.

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

/* global constant variable */
const double MY_PI = 3.141592654;


int main() {

    MY_PI *= 2;
    printf( "%lf\n", MY_PI );

    return EXIT_SUCCESS;
}
```

Et voici l'erreur de compilation produite :

```
$> gcc -o Sample Sample.c -Wall
Sample.c: In function 'main':
Sample.c:10:11: error: assignment of read-
     MY_PI *= 2;
           ^
$>
```

Le préprocesseur C