Le C est un langage à la syntaxe simple. Les seules complexités de ce langage viennent du fait qu'il agit de manière proche de la machine.

Pourtant, une partie des syntaxes autorisées par le C n'est pratiquement jamais enseignée. Attaquons-nous à ces cas mystérieux ! 

> Pour comprendre cet article, il est nécessaire d'avoir des bases dans un langage ayant une syntaxe et un fonctionnement proche du C.

Les opérateurs inusités

Il existe beaucoup d'opérateurs dans le langage C. Ici, on va parler particulièrement de deux d'entre eux qui ne sont pratiquement jamais utilisés.

L'opérateur virgule

Le premier est l'opérateur virgule. En C la virgule sert à séparer, factoriser les éléments d'une définition ou séparer les éléments d'une fonction. En bref, c'est un élément de ponctuation. Mais pas seulement ! C'est également un opérateur.

L'instruction suivante, bien qu'inutile est tout à fait valide :

```
1 printf("%d", (5,3) );
```

Elle affiche 3. L'opérateur `,` sert à juxtaposer des expressions. La valeur de l'expression complète est égale à la valeur de la dernière expression.

Cet opérateur se révèle très utile dans une boucle `for` pour multiplier les itérations. Par exemple pour incrémenter `i` et décrémenter `j` dans la même itération d'une boucle `for` on peut faire :

```
for( ; i < j ; i++, j-- ) {
 // [...]
}
```

Ou encore, dans de petits `if` pour les simplifier :

```
if( argc > 2 && argv[2][0] == '0' )
    action = 4, color = false;
```

Ici, on assigne `action` et `color`. Normalement pour faire 2 assignations, on aurait dû mettre des accolades au `if`.

On peut également s'en servir pour retirer des parenthèses.

```
while( c = getchar(), c != EOF && c != '\n' ) {
  // [...]
}
// Est strictement équivalent à :
while( (c = getchar()) != EOF && c != '\n' ) {
  // [...]
}
```

Mais surtout, ne pas abuser de cet opérateur ! On peut, de manière assez rapide, obtenir des choses illisibles. Cette remarque est d'ailleurs également valide pour le prochain opérateur !

L'opérateur ternaire

Le ternaire pour les intimes. Le seul opérateur du langage C qui prenne 3 opérandes. Il sert à simplifier des expressions conditionnelles.

Par exemple pour afficher le minimum de deux nombres, sans le ternaire, on ferait :

```
if (a < b)
    printf("%d", a);
else
    printf("%d", b);
```

Ou avec une variable temporaire :

```
int min = a;
if( b < a)
  min = b;
printf("%d", min);
```

Alors qu'avec les ternaires on fait simplement :

```
printf("%d", a<b ? a : b);
```

Grâce au ternaire, on a économisé une répétition ainsi que quelques lignes. Mais surtout on a gagné en lisibilité. Quand on sait en lire une…

Pour lire une expression ternaire, il faut la découper en 3 parties :

```
expression_1 ? expression_2 : expression_3
```

La valeur d'une expression ternaire est `expression_2` si la valeur de `expression_1` est évaluée à vrai et `expression_3` sinon.

En somme, cela simplifie un peu la lecture pour de courtes expressions. L'expression `a<b ? a : b` se lit « Si `a` est inférieur à `b` alors `a` sinon `b` ».

J'insiste encore sur le fait que cet opérateur, s'il est mal utilisé, peut nuire à la lisibilité du code. Notez que l'on peut très bien utiliser une expression

ternaire comme opérande d'une autre expression ternaire :

```
printf("%d", a<b ? a<c ? a : b<c ? b : c : b < c ? b : c);
```

Désormais, on prend le minimum de trois nombres. C'est aéré, mais impossible à suivre. Le plus lisible est d'utiliser une macro :

```
#define MIN(a,b) ((a) < (b) ? (a) : (b))

printf("%d", MIN(a,MIN(b,c)));
```

Et voilà ! Deux opérateurs qui vont désormais gagner un peu d'intérêt. Et être mieux compris ! Puis d'ailleurs, même les opérateurs que l'on connait déjà, on n'en maîtrise pas forcément la syntaxe.

L'accès à un tableau

On nous a toujours appris que pour afficher le 3ème élément d'un tableau, on faisait :

```
int tab[5] = {0, 1, 2, 3, 4, 5};
printf("%d", tab[2]);
```

2 et non 3, car un tableau en C commence à 0. Si un tableau commence à 0, c'est une histoire d'adresse [1]. L'adresse du tableau est en fait l'adresse du premier élément du tableau. Et par arithmétique des pointeurs, l'adresse du 3ème élément est `tab+2`.

Donc on aurait très bien pu écrire :

```
printf("%d", *(tab+2));
```

Puis comme l'addition est commutative, `tab+2` ou `2+tab` sont équivalents. En fait, on aurait même pu faire :

```
printf("%d", 2[tab]);
```

C'est tout à fait valide. Et pour cause : la syntaxe `E[F]` est strictement équivalente à `*((E)+(F))`. Du coup, le titre de cette section est un peu trompeur. Cet opérateur n'a pas grand-chose à voir avec les tableaux en fait. C'est du sucre syntaxique pour cacher l'arithmétique des pointeurs. [2]

Par exemple pour afficher le caractère `=` pour vrai, `!` pour faux et `~` pour aucun des deux. On pourrait faire :

```
if( is_good == 1 )
  printf("%c", '=');
else if( is_good == 0 )
  printf("%c", '!');
else
  printf("%c", '~');
```

Mais il existe plus simple :

```
printf("%c", "!=~"[is_good]);

// Ou comme on l'a vu :
printf("%c", is_good["!=~"] ); // Affiche '!' si is_good vaut 0
                               //         '=' si is_good vaut 1
                               //         '~' si is_good vaut 2
```

En somme, tout le monde écrit `tab[3]` et pas `3[tab]`. Du coup, il n'y a aucun intérêt à écrire `3[tab]`. Mais c'est toujours bon de savoir que ça existe.

---

1. C'est un peu plus compliqué que cela. À l'époque où il a fallu choisir si on commençait un tableau à l'indice 0 ou 1, le temps de compilation d'un programme avait une importance particulière. Il a été décidé de commencer à 0 afin de ne pas perdre de temps à la compilation en translation d'indices. L'article Citation Needed de Mike Hoye explique cela bien mieux que moi. ↩
2. Pour la petite histoire l'opérateur `[]` est hérité du langage B où le concept de tableau n'existait pas comme en C. Un tableau (ou vecteur comme on l'appelait à l'époque) n'était que l'adresse du premier élément d'une suite d'octets. Seul l'arithmétique des pointeurs permettait d'accéder à tous les éléments d'un tableau, c'est une évolution comparé au BCPL, l'ancêtre du B, qui utilisait la syntaxe `V!4` pour accéder au cinquième élément d'un tableau, mais je m'égare… ↩

L'initialisation

L'initialisation, c'est quelque chose que l'on maîtrise en C. C'est le fait de donner une valeur à une variable lors de sa déclaration. En gros, on définit sa valeur.

Pour un tableau [1]:

```
1 int tab[10] = {0};
2 tab[2] = 5;
```

À la première ligne, on initialise le tableau avec des 0 car, lorsqu'on initialise un tableau toute valeur non spécifiée est par défaut 0. La ligne suivante est une affectation et non une initialisation.

Si on veut seulement initialiser le troisième élément, puisque l'initialisation d'un tableau se fait suivant l'ordre de ses valeurs, on devrait écrire :

```
1 int tab[10] = {0, 0, 5};
```

Mais en réalité, il existe une autre syntaxe qui permet de faire plus simple :

```
int tab[10] = {[2] = 5};
```

On dit simplement que la troisième case vaut 5. Le reste est par défaut 0. Une syntaxe équivalente existe d'ailleurs pour les structures et les unions.

> Pour l'exemple, on va prendre une structure `point` que je vais utiliser plusieurs fois dans ce billet, idem pour la structure `message`.

On peut ainsi initialiser un point en utilisant ses composantes.

```
typedef struct point {
  int x,y;
} point;


point A = {.x = 1, .y = 2};
```

Ici, il n'y avait pas d'ambigüité. Mais pour une structure plus complexe, cette syntaxe est vraiment avantageuse.

Tenez :

```
typedef struct message {
   char src[20], dst[20], msg[200];
} message;

// [...]

message to_send = {.src="", .dst="23:12:23", .msg="Code 10"};

// Est bien plus clair que :

message to_send = {"", "23:12:23", "Code 10"};
```

```
12
13 // D'ailleurs, je ne l'ai pas fait mais avec cette syntaxe pas
14 // des champs de la structure
15
16 message to_send = { .msg="Code 10", .dst="23:12:23", .src=""};
17
18 // Et aussi puisque tout champ d'une structure est inialisé à
19 // On peut également omettre src.
20
21 message to_send = { .dst="23:12:23", .msg="Code 10"};
```

Avec ces syntaxes on peut également alourdir le code, mais généralement, on gagne en lisibilité. Parfois, ces syntaxes sont très judicieusement utilisées ! Comme ici, dans ce décodeur de base64 :

```
1  static int b64_d[] = {
2     ['A'] =  0, ['B'] =  1, ['C'] =  2, ['D'] =  3, ['E'] =
3     ['F'] =  5, ['G'] =  6, ['H'] =  7, ['I'] =  8, ['J'] =
4     ['K'] = 10, ['L'] = 11, ['M'] = 12, ['N'] = 13, ['O'] =
5     ['P'] = 15, ['Q'] = 16, ['R'] = 17, ['S'] = 18, ['T'] =
6     ['U'] = 20, ['V'] = 21, ['W'] = 22, ['X'] = 23, ['Y'] =
7     ['Z'] = 25, ['a'] = 26, ['b'] = 27, ['c'] = 28, ['d'] =
8     ['e'] = 30, ['f'] = 31, ['g'] = 32, ['h'] = 33, ['i'] =
9     ['j'] = 35, ['k'] = 36, ['l'] = 37, ['m'] = 38, ['n'] =
10    ['o'] = 40, ['p'] = 41, ['q'] = 42, ['r'] = 43, ['s'] =
11    ['t'] = 45, ['u'] = 46, ['v'] = 47, ['w'] = 48, ['x'] =
12    ['y'] = 50, ['z'] = 51, ['0'] = 52, ['1'] = 53, ['2'] =
13    ['3'] = 55, ['4'] = 56, ['5'] = 57, ['6'] = 58, ['7'] =
14    ['8'] = 60, ['9'] = 61, ['+'] = 62, ['/'] = 63, ['='] =
15 };
```

Taurre

---

1. `{0}` peut également être utilisé pour un nombre. Toute valeur initialisant une variable simple (pointeur, nombre, …) peut optionnellement prendre des accolades.

```
1 int a = {11};
2 float pi = {3.1415926};
```

```
3 char* s = {"unicorn"};
```

Ce qui caractérise le tableau du coup est surtout le fait d'utiliser des virgules entre les accolades. ↩

Les expressions littéralement composées

Puisque nous parlons des tableaux. Il existe une syntaxe simple pour utiliser des tableaux à usage unique.

Je voudrais utiliser ce tableau :

```
1 int tab[5] = {5, 4, 5, 2, 1};
2 printf("%d", tab[i]); // Avec i égale à quelque chose >=0 et <5
```

Cependant, je ne l'utilise qu'une seule fois ce tableau … C'est un peu dérangeant d'avoir à utiliser un identificateur juste pour ça.

Eh bien je peux faire ceci :

```
1 printf("%d", ((int[]){5,4,5,2,1}) [i] ); // Avec i égale à quel
```

Ce n'est pas super lisible pour le coup. Mais il existe plein de cas où cette syntaxe est très utile. Par exemple avec une structure :

```
1 // Pour envoyer notre message :
2 send_msg( (message){ .dst="192.168.11.1", .msg="Code 11"} );
3
4 // Pour afficher la distance entre deux points
5 printf("%d", distance( (point){1, 2}, (point){2, 3} )  );
6
7
```

```
8  // Ou encore sous Linux, en programmation système
   execvp( "bash" , (char*[]){"bash", "-c", "ls", NULL} );
```

On appelle ces expressions des *compound literals* (ou littéraux agrégats si l'on s'essaye à traduire).

Introduction aux VLAs

Les tableaux à taille variable (ou *Variable Length Arrays* en anglais, soit VLAs) sont des tableaux dont la taille n'est connue qu'à l'exécution. Si vous n'avez jamais entendu parler des VLAs, ceci devrait vous choquer :

```
1  int n = 11;
2
3  int tab[n];
4
5  for(int i = 0 ; i < n ; i++)
6    tab[i] = 0;
```

Ce code, bien que valide, a dû être réprimandé par de nombreux professeurs. En effet, on nous apprend qu'un tableau doit avoir une taille connue à la compilation. *Eh* bien, les VLAs constituent l'exception. Apparu avec la norme C99, les VLAs jouissent d'une mauvaise réputation. À cela, il existe plusieurs raisons que je ne détaillerais pas ici [1]. Je vais simplement parler des comportements non-intuitifs introduits avec les VLAs. Mais tout d'abord, voyons ce qu'est et comment se servir d'un tableau à taille variable.

Les VLAs se définissent avec la même syntaxe qu'un tableau classique. La seule différence est que la taille du tableau est une expression entière non constante.

```
1  int n = 50;
2  int tab[n];
3
```

```
4 double tab2[2*n];
5
6 unsigned int tab[foo()]; // avec foo une fonction définie aille
```

Un VLA ne peut pas être initialisé, de plus, il ne peut être déclaré `static` . C'est-à-dire que ces deux déclarations sont incorrectes :

```
1 int n = 30;
2
3 int tab[n] = {0};
4 static tab2[n];
```

Dans une fonction, on pourrait utiliser :

```
1 void bar(int n, int tab[n]) {
2
3 }
```

Cependant, la taille de la première dimension d'un tableau n'a pas beaucoup d'importance, un tableau étant implicitement convertit en un pointeur vers son premier élément. En revanche, pour un tableau à deux dimensions, la taille de la deuxième dimension *doit* être spécifiée :

```
1 void foo( int n, int m, int tab[][m]) {
2
3 }
```

À noter qu'il est possible d'utiliser le caractère `*` (encore une utilisation de plus) en lieu et place de la taille d'une ou plusieurs dimensions d'un VLA, mais *uniquement* au sein d'un prototype.

```
1 void foo(int, int, int[][*]);
```

Bien, après cette courte introduction aux VLAs, passons aux cas qui nous intéressent. Pour être précis, les bizarreries et excentricités que les VLAs

ont introduits.

---

1. Je peux néanmoins vous donnez deux références "Is it safe to use variable-length arrays?" de stack overflow et cet article : "The Linux Kernel Is Now VLA-Free" . ↩

L'exception des VLAs

Le comportement déviant le plus connu des VLAs est leur rapport à `sizeof` .

`sizeof` est un opérateur unaire qui permet de retrouver la taille d'un type à partir d'une expression ou du nom d'un type entouré de parenthèses.

```c
/*  Le fonctionnement de l'opérateur sizeof par des exemples */
float a;
size_t b = 0;

printf("%zu", sizeof(char)); // Affiche 1
printf("%zu", sizeof(int));  // Affiche 4
printf("%zu", sizeof a);     // Affiche 4
printf("%zu", sizeof(a*2.)); // Affiche 8
printf("%zu", sizeof b++);   // Affiche 8
```

Le premier résultat n'est pas très surprenant, la taille d'un `char` est défini à 1 *byte* et `sizeof(char)` doit retourner 1. Le deuxième résultat est la taille d'un `int` [1]. Le troisième résultat est la taille d'un `float` [1] . Le quatrième est la taille d'un `double` [1] qui est le type de l'expression `a*2.` [2]. Le dernier résultat est la taille d'un `size_t` [1], `size_t` étant le type de l'expression `b++` .

Ici, je ne me suis pas intéressé à la valeur des expressions et pour cause, `sizeof` ne s'y intéresse pas non plus. Ces valeurs sont déterminées à la compilation. Les opérations que l'on retrouve dans l'expression passé à

`sizeof` ne sont pas effectuées. Puisque l'expression doit être valide, son type doit être déterminé à la compilation. Le résultat de `sizeof` étant alors connu à la compilation, il n'y avait aucune raison d'exécuter l'expression.

```
1 int n = 5;
2 printf("%zu", sizeof(char[++n])); // Affiche 6
```

Arf ! Les VLAs rajoutent leur grain de sel. Dans le type `int[++n]`, `++n` est une expression non constante. Donc le tableau est un tableau à taille variable. Pour connaitre la taille totale du tableau, il est nécessaire d'exécuter l'expression entre crochet. Ainsi `n` vaut désormais 6 et `sizeof` nous indique qu'un VLA de `char` déclaré avec cette expression aurait eu pour taille `6`.

Ce n'est que peu intuitif puisque les VLAs ont ici introduit une *exception à la règle* qui est de ne pas exécuter l'expression passée à `sizeof`.

Un autre comportement bizarre introduit par les VLAs est l'exécution des expressions liées à la taille d'un VLA dans la définition d'une fonction. Ainsi :

```
1 int foo( char tab[printf("bar")] ) {
2    printf("%zu", sizeof tab);
3 }
```

En supposant que les affichages ne causent pas d'erreur, appeler cette fonction affichera `bar3`. L'instruction `printf("bar")` est évaluée et ensuite seulement le corps de la fonction est exécuté.

Il est à noter qu'il existe d'autres exceptions induites pas la standardisation des VLAs comme l'impossibilité d'allouer les VLAs de manière statique (assez logique), ou l'impossibilité d'utiliser des VLAs dans une structure (GNU GCC le support tout de même). Et même certains branchements conditionnels sont interdits lorsque l'on utilise un VLA.

---

1. Sur ma machine ↩
2. Ici, l'implémentation des nombres flottants suit la norme IEE 754, où la taille d'un nombre flottant simple est de 4 octets et double précision est de 8 octets. `2.` est de type `double`, l'expression `2.*a` est donc du même type que le plus grand des deux types de ses deux opérandes, ici `double`. ↩

Un tableau flexible

Vous n'avez peut-être jamais entendu parler des « *flexible arrays member* ». C'est normal, ceux-ci répondent à une problématique très précise et peu courante.

L'objectif est d'allouer une structure mais dont un champ (un tableau) est de taille inconnue à la compilation et le tout sur un espace contiguë [1].

Ici, pas de VLAs, car comme on l'a vu, ceux-ci sont interdits en tant que champs de structure. L'allocation dynamique s'impose.
On pourrait vouloir faire :

```
1  struct foo {
2    int* tab;
3  };
```

Et l'utiliser comme ceci :

```
1    struct foo* contigue = malloc( sizeof(struct foo) );
2    if (contigue) {
3      contigue->tab = malloc( N * sizeof *contigue->tab );
4      if (contigue->tab) {
5        contigue->tab[0] = 11;
6      }
7    }
```

Mais ici, le tableau n'a aucune raison d'être contiguë à la structure. Ce qui aura pour conséquence qu'en cas de copie de la structure, la valeur du champ `tab` sera la même pour la copie et l'origine. Pour éviter cela, il faudra copier la structure, réallouer le tableau et le recopier. Voyons une autre méthode.

```
1  struct foo {
2    /* ... Au moins un autre champ car le standard l'impose. */
3    int flexiTab[];
4  };
```

Ici, le champ `flexiTab` est un tableau membre flexible. Un tel tableau doit être le dernier élément de la structure et ne pas spécifier de taille [2]. On l'utilise comme ceci :

```
1    struct foo* contigue = malloc( sizeof(struct foo) + N * sizec
2    if (contigue) {
3      flexiTab[0] = 11;
4    }
```

Cette syntaxe répond autant à un besoin de portabilité sur les architectures imposant un alignement particulier (le tableau est contigu à la structure) qu'au besoin de faire apparaître un lien sémantique entre le tableau et la structure (le tableau appartient à la structure).

---

1. On peut vouloir que cet espace soit contiguë pour plusieurs raisons. Une de ces raisons est pour optimiser l'utilisation du cache processeur. Également, la gestion des couches réseaux se portent assez bien à l'utilisation de tableaux flexibles. ↩
2. Avant la spécification des « flexible array member » en C99, il était courant d'utiliser des tableaux de taille un pour reproduire le concept. ↩

Une histoire d'étiquettes

En C, s'il y a un truc dont on ne doit pas parler, ce sont bien des *étiquettes* . On les utilise avec la structure de contrôle `goto` ! L'interdite !

Pour les cacher, on remplace les `goto` par des structures de contrôles adaptées et plus claires comme `break` ou `continue` . De manière à ne jamais avoir à utiliser `goto` .

Du coup, on n'apprend jamais ce qu'est une étiquette…

Voici comment on utilise `goto` et une étiquette :

```
1  goto end;
2
3
4  end: return 0;
5  }
```

En gros, une étiquette est un nom que l'on donne à une instruction.

Eh bien on en utilise des étiquettes ! Dans les `switch` !

```
1  switch( action ) {
2      case 0:
3          do_action0();
4      case 1:
5          do_action1();
6          break;
7      case 2:
8          do_action2();
9      break;
10     default:
11         do_action3();
12 }
```

Ici, les `case` et le `default` sont en fait des étiquettes ! Comme pour les `goto` . Sauf qu'elles sont pour les `switch` , et inutilisables par les `goto` …

Pourquoi tu nous parles de ça ?

Déjà, c'est bien de savoir que ça s'appelle une étiquette. Ensuite, parce que je vais vous parlez d'un classique. Le dispositif de *Duff* .

C'est une sorte de boucle déroulée optimisée. Le but est de réduire le nombre de vérifications de fin de boucle (ainsi que le nombre de décrémentations).

Voici la version historique écrite par Tom Duff :

```
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

Peu importe ce que signifie `register` . Aussi, `to` est un pointeur particulier, mais ce n'est pas vraiment important.

Ici, ce dont je veux vous parler, c'est de cette boucle `do-while` en plein milieu d'un `switch` .

Le test que l'on cherche à effectuer le moins possible est `--n > 0` .

En temps normal, `n` serait en fait `count` . Et on devrait faire le test `count` fois. De même pour sa décrémentation.

C'est-à-dire :

```c
while( count-- > 0 )
    *to = *from++;
```

En divisant par 8 (nombre arbitraire) on divise également par 8 le nombre de tests et de décrémentations. Cependant, si `count` n'est pas divisible par 8, on a un problème, on ne fait pas toutes les instructions. Ce serait bien de pouvoir sauter directement à la 2ème instruction, si on a seulement 6 instructions restantes.

Et c'est là que les étiquettes peuvent nous aider ! Grâce au `switch` on peut sauter directement à la bonne instruction.

Il suffit d'étiqueter chaque instruction avec le nombre d'instructions qu'il reste à faire dans la boucle. Puis de sauter dans la boucle avec la structure de contrôle `switch` sur le reste d'instructions à réaliser.

Ensuite, on exécute nos paquets de 8 instructions normalement.

Il est très rare d'avoir à utiliser ce type d'astuce. D'ailleurs, c'est une optimisation d'un autre temps. Mais comme je voulais vous parler de syntaxe, il était nécessaire de parler des étiquettes.

Les nombres complexes

Encore une fois nous allons voir une syntaxe introduite en C99. Plus exactement, ce sont 3 types qui ont été introduits, ceux-ci correspondent aux nombres complexes. Le type d'un nombre complexe est `double _Complex` (les deux autres types suivent le même schéma, afin de ne pas me répéter, je vais me concentrer sur le type `double`).

Ainsi en C, il est possible de déclarer un nombre complexe comme ceci :

```c
double complex point = 2 + 3 * I;
```

Ici, on retrouve les macros spéciales `complex` et `I` (définie dans l'en-tête `<complex.h>`). Le premier sert à créer un type complexe alors que le second sert à définir la partie imaginaire d'un nombre complexe.

En mémoire une variable complexe occupe autant d'espace que 2 fois le type réel sur lequel elle est basée. On se sert d'une variable complexe comme d'une variable normale. L'arithmétique y est intuitive puisque basées sur celle des réels. Il est à noter qu'il est recommandé d'utiliser le macro `CMPLX` pour initialiser un nombre complexe :

```
1 double complex cplx = CMPLX(2, 3);
```

Pour une meilleure gestion des cas où la partie imaginaire (celle multipliée par `I` donc) serait `NAN`, `INFINITY` ou encore plus ou moins 0.

L'en-tête `<complex.h>` nous offre une manière réellement simple d'utiliser des nombres imaginaires. En effet, de nombreuses fonctions courantes de manipulations des nombres imaginaires y sont disponibles.

Les macros génériques

Il existe un moyen en C d'avoir des macros qui soient définies différemment en fonction du type de l'un de ses arguments. Cette syntaxe est cependant « nouvelle » puisqu'elle date du standard C11.

Cette généricité s'obtient avec les sélections génériques basées sur la syntaxe `_Generic ( /* ... */ )`

Pour comprendre la syntaxe, voyons un exemple simpliste :

```
1 #include <stdio.h>
2 #include <limits.h>
3
4 #define MAXIMUM_OF(x) _Generic ((x), \
5                                 char: CHAR_MAX, \
6                                 int:  INT_MAX   \
```

```
                                    int: INT_MAX,   \
                                    long: LONG_MAX  \
                                    )

int main(int argc, char* argv[]) {
    int i = 0;
    long l = 0;
    char c = 0;

    printf("%i\n", MAXIMUM_OF(i));
    printf("%d\n", MAXIMUM_OF(c));
    printf("%ld\n", MAXIMUM_OF(l));
    return 0;
}
```

Ici, on affiche le maximum que peut stocker chacun des types que nous utilisons. C'est quelque chose qui n'aurait pas été possible sans l'utilisation de ce nouveau mot-clé `_Generic`. Pour utiliser cette syntaxe, on utilise le mot clé `_Generic` auquel on passe 2 paramètres. Le premier est une expression dont le type va influencer l'expression finalement exécutée. Le deuxième est une suite d'association de type et d'expression (type : expression) dont les associations sont séparées par des virgules. Au final, seule l'expression désignée par le type de la première expression est finalement évaluée.

Un exemple réel d'utilisation pourrait être :

```
int powInt(int,int);

#define POW(x,y) _Generic ((y), double: pow, float: powf, long
```

Il n'y a plus grand-chose à dire si ce n'est qu'il est possible d'avoir dans la liste des types le mot `default` qui correspondra alors à tous les types non-cités. Ainsi une définition plus propre de la macro `POW` de tout à l'heure pourrait être :

```
int powIntuInt(int a, unsigned int b);
double powIntInt(int a, int b);
double powFltInt(double a,int b) { return pow (a,b); }
```

```
4  double powfFltInt(float a,int b) { return powf(a,b); }
5  double powlFltInt(long double a,int b) { return powl(a,b); }
6
7
8  #define POWINT(x) _Generic((x), double: powFltInt,          \
9                                  float : powfFltInt,         \
10                                 long double: powlFltInt,    \
11                                 unsigned int: powIntuInt,   \
12                                 default: powIntInt)
13 #define POW(x,y) _Generic ((y), double: pow, float: powf, long
```

Les caractères trop spéciaux

On va remonter encore dans le temps. Je vais vous citer une dernière chose. Un temps où tous les caractères n'étaient pas aussi accessibles que de nos jours sur autant de types de claviers.

Les claviers n'avaient pas forcément de touches de composition. Ainsi, il était impossible de taper le caractère `#`.

On pouvait alors remplacer le caractère `#` par la séquence `??=`. Et pour chaque caractère n'étant pas sur le clavier et utiliser en langage C, il existait une séquence à base de `??` que l'on nomme trigraphe. Une autre version à base de 2 caractères plus lisible se nomme les digraphes.

Voici un tableau récapitulatif des séquences de trigraphes, de digraphes et de leur représentation en caractère.

Caractère	Digraphe	Trigraphe
`#`	`%:`	`??=`
`[`	`<:`	`??(`
`]`	`:>`	`??)`
`{`	`<%`	`??<`

Caractère	Digraphe	Trigraphe
`}`	`%>`	`??>`
`\`		`??/`
`^`		`??'`
`|`		`??!`
`~`		`??-`
`##`	`%:%:`	

La différence **principale** entre les digraphes et les trigraphes est dans une chaîne de caractère :

```
puts("??= est un croisillon");
puts("%:");
```

Ces mécanismes du Moyen Âge sont encore valides de nos jours en C. Du coup, cette ligne de code est tout à fait valide :

```
??=define FIRST tab<:0]
```

La seule utilité de cette syntaxe de nos jours est d'obfusquer un code source très facilement. Une combinaison d'un ternaire avec un trigraphe et un digraphe et vous avez un code absolument illisible. 

```
printf("%d", a ?5??((tab):>:0);
```

À ne jamais utiliser dans un code sérieux donc.

Voilà, c'est fini, j'espère que vous avez appris quelque chose de ce billet. Surtout n'oubliez pas d'utiliser ces syntaxes avec parcimonie.

Je tiens tout particulièrement à remercier Taurre pour avoir validé cet article, mais également pour sa pédagogie sur les forums depuis des années, ainsi que blo yhg pour sa relecture attentive.

À noter que vous pouvez (re)découvrir de nombreux codes abusant de la syntaxe du langage C aux IOCCC .