Taurre

Geste de savoir

La norme C23 est dans les cartons

lundi 16 septembre 2024

Table des matières

Introduction

Depuis quelque temps à présent[1], le brouillon de la future norme C a été finalisé et le document suit son court au sein de l' ISO[2] en vue de sa publication. Entre-temps, les compilateurs et diverses implémentations de la bibliothèque standard ont commencé à avancer dans leur support de cette future norme. C'est l'occasion de voir ce que nous réserve la future mouture de notre humble langage cinquantenaire.

i

Ce billet n'a pas pour vocation d'être exhaustif quant aux changements apportés, la liste complète des changements est exposée dans les premières pages du brouillon de la norme.

!

À l'heure où ces lignes sont écrites, la norme C23 n'est pas encore complètement supportée, ni par les compilateurs, ni par les différentes implémentations de la bibliothèque standard. En conséquence, certains codes présentés ne peuvent actuellement pas être compilés (ils sont marqués par un encart orange) et leur comportement est uniquement déduit de la lecture de la norme.

1. Orthographe alternative pour certains mot-clés

La norme C11 avait introduit les mot-clés `_Alignas`, `_Alignof`, `_Bool`, `_Static_assert` et `_Thread_local`. Ceux-ci sont toujours supportés, mais deviennent obsolètes (ils seront [peut-être] supprimés dans une future mouture de la norme) et sont remplacés par : `alignas`, `alignof`, `bool`, `static_assert` et `thread_local`[1].

2. Constantes true et false

La norme C11 avait introduit deux macroconstantes : `true` et `false` définies dans l'en-tête `<stdbool.h>` et valant respectivement `1` et `0`[1].

1. https://mailman.oakapple.net/pipermail/cfp-interest/2023-December/002982.html
2. https://www.iso.org/fr/standard/82075.html
1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.30 Unicode utilities <uchar.h>, al. 3, p. 407.

La norme C23 considère désormais `true` et `false` comme deux mot-clés[2], mais également comme deux constantes prédéfinies de type `bool` avec pour valeur respective `1` et `0`[3]. Le type de ces deux constantes est le changement le plus important, notamment pour les sélections génériques qui attribueront le type `bool` à ces constantes et non plus le type `int`.

```c
#include <stdio.h>

int
main(void) {
        printf("Type of true: %s\n", _Generic(true, bool: "bool",
            default: "int"));
        printf("Type of false: %s\n", _Generic(false, bool:
            "bool", default: "int"));
        return 0;
}
```

```
Type of true: bool
Type of false: bool
```

3. Constantes entières binaires et séparateurs de chiffres

3.1. Constantes entières binaires

Il est désormais possible d'utiliser des constantes entières binaires en préfixant la constante par les lettres `0b` ou `0B`[1]. Le type de la constante est déterminé de la même manière que pour les constantes entières hexadécimales ou octales[2].

```c
#include <stdio.h>

int
main(void) {
        printf("%d\n", 0b1000);
        return 0;
}
```

```
8
```

1. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 7.18 Boolean type and values <stdbool.h>, p. 287.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.1 Keywords, al. 1, p. 53.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.4.5 Predefined constants, al. 1–3, p. 66.

À noter que les fonctions des familles `printf()`[3] et `scanf()`[4] se voient également ajouter les formats `%b` et `%B`qui permettent d'écrire ou de lire un nombre binaire.

```c
#include <stdio.h>

int
main(void) {
        printf("%d: %b\n", 55, 55);
        return 0;
}
```

```
55: 110111
```

3.2. Séparateurs de chiffres

Afin d'améliorer la lisibilité des constantes entières et flottantes (il n'est tout de même pas facile de lire `0b0000101000000000` 🍊), il est désormais possible d'insérer un séparateur (`'`) entre n'importe quels chiffres composant une constante entière ou flottante[5] [6]. Il est par exemple possible de séparer chaque groupe de quatre chiffres.

```c
#include <stdio.h>

int
main(void) {
        printf("%d\n", 0b0000'1010'0000'0000);
        printf("%f\n", 3.1415'9265'3589'7932);
        return 0;
}
```

```
2560
3.141593
```

Les séparateurs ne peuvent être placés qu'*entre les chiffres composant une constante*. Ils ne peuvent pas être utilisés pour séparer la constante de son préfixe, de son suffixe, du point décimal ou du `e` de l'exposant. Ils ne peuvent pas non plus être utilisés au début ou

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.4.1 Integer constants, al. 4, p. 59.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.4.1 Integer constants, al. 6, p. 59.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.23.6.1 The fprintf function, al. 8, p. 330.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.23.6.2 The fscanf function, al. 12, p. 338.

à la fin de la constante.

4. Introduction de la constante nullptr et du type nullptr_t

Depuis ses débuts, le C dispose d'une macroconstante `NULL` (définie dans l'en-tête `<stddef.h>`) dont la valeur doit être une constante pointeur nul. Celle-ci est définie comme une expression constante entière nulle (basiquement zéro) facultativement convertie vers le type `void *` [1]. Dit autrement, cela laisse deux possibilités de valeur pour la macroconstante `NULL` : soit `0`, soit `(void *)0`.

Cependant, pour obtenir un pointeur nul, il est en plus nécessaire de convertir cette constante vers un type pointeur[1]. Dans le cas de `(void *)0`, ce n'est — dans les faits — pas nécessaire puisqu'il s'agit déjà d'une expression de type pointeur. En revanche, pour le cas de zéro, c'est primordial.

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

void
print_two_int(int *p) {
        /*
         * 0 est implicitement converti vers le type (int *).
         * Nous vérifions donc bien si p est ou non un pointeur
             nul.
         */
        if (p == 0) {
                exit(EXIT_FAILURE);
        }

        printf("%d, %d\n", p[0], p[1]);
}

int
main(void) {
        /*
         * 0 est implicitement converti vers le type (int *).
         * p est donc un pointeur nul.
         */
        int *p = 0;
        p = malloc(sizeof *p * 2);
        p[0] = 1;
        p[1] = 2;
        print_two_int(p);

```

5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.4.1 Integer constants, al. 2, p. 58.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.4.2 Floating constants, al. 3, p. 61.

```
29          /*
30           * Le compilateur sait que le premier argument de
                 print_two_int() est un (int *).
31           * Il convertit donc 0 vers le type (int *).
32           * Le paramètre p est donc un pointeur nul.
33           */
34          print_two_int(0);
35          return 0;
36  }
```

```
1  1, 2
```

Or, si cela est implicite dans pas mal de cas, par exemple en cas d'une comparaison ou d'une assignation, il y a des cas où cela pose problème et notamment le cas des fonctions à nombre variable d'arguments comme `printf()`.

En effet, le nombre d'arguments étant inconnu, le compilateur ne peut pas effectuer de conversions depuis le type de l'argument (l'expression passée lors de l'appel de fonction) vers le type du paramètre correspondant (la variable qui est initialisée à l'aide de l'argument). Dans un tel cas, le compilateur applique une règle de promotion générale : les entiers subissent la promotion intégrale et les flottants de type `float` sont convertis en `double`[2].

Or, zéro est un entier et le restera donc en application de cette règle générale. Et c'est là que le bat blesse : écrire `printf("%p\n", NULL)` peut conduire à passer un entier valant zéro alors que `printf()` attends un pointeur, ce qui constitue un comportement indéfini[3].

4.1. La constante nullptr

La norme C23 introduit la constante `nullptr`, de type `nullptr_t` (défini dans l'en-tête `<stddef.h>`), qui vient s'ajouter à la liste des constantes pointeur nul.

Le type `nullptr_t` a la même taille et les mêmes contraintes d'alignements qu'un pointeur sur `char`[4] (et, conséquemment, les mêmes qu'un pointeur sur `void`[5]) et la constante `nullptr`[6] a la même représentation que l'expression `(void *)0`[4].

Au vu de ses propriétés, la constante `nullptr` vient combler les problèmes de la macroconstantes `NULL` et peut être utilisée sans problème partout où un pointeur nul est attendu.

```
1  #include <stdio.h>
2  #include <stddef.h>
3  #include <string.h>
4
5  int
6  main(void) {
7          nullptr_t n = nullptr;
```

```c
        char *p = 0;
        void *q = (void *)0;
        printf("%zu, %zu, %zu\n", sizeof n, sizeof p, sizeof q);
        printf("%zu, %zu, %zu\n", alignof(nullptr_t),
            alignof(char *), alignof(void *));
        printf("n == p: %s\n", memcmp(&n, &p, sizeof n) == 0 ?
            "oui" : "non");
        printf("n == q: %s\n", memcmp(&n, &p, sizeof n) == 0 ?
            "oui" : "non");
        printf("n = %p, p = %p, q = %p\n", n, p, q);
        return 0;
}
```

```
8, 8, 8
8, 8, 8
n == p: oui
n == q: oui
n = (nil), p = (nil), q = (nil)
```

5. Les entiers de taille arbitraire

Un nouveau mot-clé `_BitInt` a été introduit et permet de construire un type entier de taille arbitraire sous la forme `_BitInt(N)` où `N` est la taille en *bits* (*bit* de signe inclut, s'il s'agit d'un type signé). Il est donc désormais possible de manipuler des entiers sur, par exemple, 24 *bits*. La taille en *bits* ne peut être inférieure à 2 (pour un type signé) ou 1 (pour un type non signé)[1] et ne doit pas excéder la taille du plus grand type entier supporté, types entiers étendus inclus[2].

Notez que ces types ont une propriété particulière : ils ne subissent *pas* la promotion entière (ou promotion intégrale)[3]. Pour rappelle, cette règle spécifie que si une valeur entière d'un rang inférieur à `int` ou `unsigned int` (le rang se détermine notamment suivant l'intervalle de valeurs représentables) est utilisée dans une expression, alors celle-ci est convertie vers le type `int` ou `unsigned int`.

```c
#include <stdio.h>

int
main(void) {
        unsigned _BitInt(3) a = 1;
```

1. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 6.3.2.3 Pointers, al. 3, p. 55.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.1 Keywords, al. 6, p. 75.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.5 Expressions, al. 7, p. 71.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.21.2 The nullptr_t type, al. 3, p. 313.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.2.5 Types, al. 33, p. 40.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.4.5 Predefined constants, al. 4, p. 66.

```
6           unsigned _BitInt(3) b = 2;
7           unsigned _BitInt(3) c = a + b; /* Ici, ni a, ni b ne sont
                convertis en int, le type reste _BitInt(3) */
8
9           printf("%d\n", a); /* a n'est pas converti en int et reste
                également de type _BitInt(3) */
10          return 0;
11  }
```

```
1  main.c:8:17: warning: format specifies type 'int' but the argument
       has type '_BitInt(3)' [-Wformat]
2           printf("%d\n", a + b);
3                   ~~     ^~~~
```

Dans l'exemple ci-dessus, le compilateur vous signale que le format `%d` attend un `int` et non un `_BitInt(3)` ce qui démontre bien que la promotion n'a pas eu lieu.

En revanche, les conversions arithmétiques usuelles demeurent. À ce sujet, le rang d'un type entier de taille arbitraire se détermine sur base de son nombre de *bits*[4].

```
1  #include <stdio.h>
2
3  int
4  main(void) {
5           unsigned _BitInt(3) a = 1;
6           unsigned _BitInt(3) b = 2;
7           _BitInt(48) c = 1023;
8           int d = 2;
9
10          printf("%d\n", a + d); /* a est converti en int */
11          printf("%d\n", (int)a + b); /* a et b sont convertis en
                int */
12          printf("%ld\n", c + d); /* c est converti en _BitInt(48),
                le format est donc incorrect */
13          return 0;
14  }
```

```
1  main.c:12:25: warning: format specifies type 'long' but the
       argument has type '_BitInt(48)' [-Wformat]
2           printf("%ld\n", c + d); /* c est converti en _BitInt(48),
                le format est donc incorrect */
3                   ~~~     ^~~~
4  3
5  3
```

```
6 1025
```

6. La représentation des nombres entiers

La norme C23 consacre la représentation en complément à deux comme la *seule* représentation des nombres entiers signés[1]. Jusqu'à présent, cette représentation n'était garantie que pour les entiers de taille fixe de l'en-tête `<stdint.h>`[2] (`intXX_t`, introduit en C99). Pour les autres entiers signés, les représentations en signe et magnitude ou en complément à un restaient possible (même si, dans les faits, ces représentations ne sont plus utilisées).

Cette décision à plusieurs implications, la première étant que l'asymétrie entre les bornes minimales et maximales des types signés est désormais garantie par la norme. Par exemple, les bornes du type `int` garanties par la norme sont dorénavant -32768 et 32767[3] là où il s'agissait de -32767 et 32767 auparavant.

La norme introduit également de nouvelles constantes dans l'en-tête `<limits.h>`[5] pour chaque type entier spécifiant leur nombre minimum garanti de *bits* hors *bits* de *padding*, soit le nombre de *bits* de valeur pour un type non signé et le nombre de *bits* de valeur et le *bit* de signe pour un type signé[4].

Ce nombre de *bits* hors *bits* de *padding* est appelé la « grandeur » (*width* en anglais) d'un type par la norme[4].

> *i* Pour rappel, un type entier est composé de *bits* de valeur et, potentiellement, de *bits* de *padding* (sauf le type `char`, qui ne peut pas avoir de *bits* de *padding*[4]). À ceux-ci s'ajoute un *bit* de signe pour les entiers signés. Il est ainsi parfaitement possible d'avoir un type `unsigned int` composé de 32 *bits*, dont 16 *bits* de valeur et 16 *bits* de *padding*. Ceci se traduit par un intervalle de valeur représentable compris entre 0 et 65535.

Ces constantes ont la forme `TYPE_WIDTH` et suivent la même convention que les constantes `TYPE_MAX`[5] [3].

Type	Constante	Minimum garanti
`bool`	`BOOL_WIDTH`	1
`signed char`	`SCHAR_WIDTH`	`CHAR_BIT`
`unsigned char`	`UCHAR_WIDTH`	`CHAR_BIT`
`char`	`CHAR_WIDTH`	`CHAR_BIT`

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.2 Type specifiers, al. 4, p. 102.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 5.2.4.2.1 Characteristics of integer types <limits.h>, al. 1, p. 22.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.3.1.1 Boolean, characters, and integers, al. 2, p. 45.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.3.1.1 Boolean, characters, and integers, al. 1, p. 44–45.

`short`	`SHRT_WIDTH`	`USHRT_WIDTH`
`unsigned short`	`USHRT_WIDTH`	16
`int`	`INT_WIDTH`	`UINT_WIDTH`
`unsigned int`	`UINT_WIDTH`	16
`long`	`LONG_WIDTH`	`ULONG_WIDTH`
`unsigned long`	`ULONG_WIDTH`	32
`long long`	`LLONG_WIDTH`	`ULLONG_WIDTH`
`unsigned long long`	`ULLONG_WIDTH`	64

7. De vraies constantes avec « constexpr »

La norme C23 ajoute le mot-clé `constexpr`. Celui-ci peut-être appliqué à une variable pour spécifier qu'elle peut être traitée comme une expression constante[1] (d'où le nom `constexpr`).

Jusqu'à présent, le C ne disposait pas de moyen pour définir des constantes en dehors des macroconstantes. Même la combinaison du qualificateur `const` et de la classe de stockage statique ne permettait pas d'atteindre cet objectif.

```
1 static const taille = 10;
2 int tableau[taille] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; /*
      Incorrect */
```

Le code ci-dessus est invalide car il définit un tableau de longueur variable (*variable-length array* ou VLA) et que ces derniers ne peuvent être initialisés lors de leur définition[5].

Avec le mot-clé `constexpr`, la variable `taille` sera considérée comme une expression constante et le tableau `tableau` sera de classe de stockage automatique.

```
1 constexpr int taille = 10;
2 int tableau[taille] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; /* Ok */
```

Ce mot-clé vient toutefois avec son lot de restrictions, notamment[2] :

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.2.6.2 Integer types, al. 2, p. 41–42.
2. ISO/IEC 9899:TC3, doc. N1256 , 07/09/2007, § 7.18.1.1 Exact-width integer types, al. 3, p. 256.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, Annexe E : Implementation limits, al. 1–3, p. 503.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.2.6.2 Integer types, al. 1–2, p. 41–42.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 5.2.4.2.1 Characteristics of integer types <limits.h>, al. 1, p. 21–22.

— Il ne peut être utilisé que lors de la *définition* d'une variable ;
— La définition doit comporter une initialisation ;
— Si la variable est de type :
— — entier, alors l'expression utilisée pour l'initialiser doit être une expression constante entière,
— — pointeur, alors l'expression utilisée pour l'initialiser doit être une constante pointeur nul,
— — flottant, alors l'expression utilisée pour l'initialiser doit être une expression constante entière ou flottante.
— La valeur assignée doit être représentable par le type de destination, aucun changement de valeur ne doit être nécessaire.

```
1  constexpr int i; /* Invalide, initialisation manquante */
2  constexpr unsigned int u = -1; /* Invalide, la valeur n'est pas
       représentable */
3  constexpr int d = 10.1; /* Invalide, il ne s'agit pas d'une
       constante entière */
4
5  constexpr void *p = (void *)1; /* Invalide, il ne s'agit pas d'une
       constante pointeur nul */
6  constexpr char *q = nullptr; /* Ok */
7  constexpr int *r = 0; /* Ok */
8
9  constexpr double e = LONG_MAX; /* Invalide, la valeur n'est pas
       représentable */
10 constexpr double f = 0.1 + 3.0 - 0.7; /* Potentiellement invalide
       */
11 constexpr double g = (double)(0.1 + 3.0 - 0.7); /* Ok */
12 constexpr double h = 5; /* Ok */
13
14 constexpr char s[] = "Un, deux, trois"; /* Ok */
15 constexpr unsigned char t[] = "\xFF"; /* Potentielement invalide */
```

Dans le cas de l'expression `0.1 + 3.0 - 0.7`, le problème vient du fait que les calculs flottants peuvent avoir lieu avec une précision plus grande que celle induite par leur type[3] [4]. Dit autrement, même si les opérandes sont de type `double`, les calculs pourraient s'effectuer avec la précision du type `long double`, pour ensuite être convertis vers le type `double`. Cette conversion peut induire un changement de valeur.

Quant au tableau `t`, le problème tient au fait qu'une chaîne de caractères constante est un tableau de `char`, type qui peut être signé ou non signé. Dans le cas où le type `char` est signé, la définition revient à écrire `unsigned char t[] = { -1, 0 };` ce qui pose le même problème que pour la variable `u` : `-1` n'est pas représentable par un `unsigned char`, un changement de

valeur est nécessaire.

8. Une vraie initialisation à zéro

L'initialisation des variables en C paraît simple, mais elle regorge de petites subtilités qui peuvent conduire à des situations non désirées. Commençons par rappeler la règle de base, qui ne change pas : une variable non initialisée, sauf si elle est de classe de stockage `static` ou `thread_local`, a une valeur indéterminée[1].

```
{
        int n; /* Valeur inconnue */
}
```

Dans le cas des variables de classe de stockage `static` ou `thread_local`, la norme garantit qu'elles sont « initialisées à zéro », plus précisément[2] :

— Un pointeur sera un pointeur nul ;
— Un nombre (réel ou entier) sera à zéro (positif ou non signé) ;
— Une structure verra tous ses membres mis à zéro, mais également tous ses éventuels *bytes* de *padding* ;
— Une union verra son *premier membre* mis à zéro, mais également tous ses éventuels *bytes* de *padding* ;
— Un tableau verra tous ses éléments mis à zéro.

Cette initialisation est appelée « initialisation par défaut » (*default initialization*) par la norme.

```
struct s {
        int x;
        double y;
};

union u {
        int x;
        double y;
};

static void *p; /* Pointeur nul */
static double n; /* 0.0 */
```

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.1 Storage-class specifiers, al. 15, p. 99.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.1 Storage-class specifiers, al. 5–6, p. 98.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.3.1.8 Usual arithmetic conversions, al. 2, p. 48.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 5.2.4.2.2 Characteristics of floating types <float.h>, al. 19, p. 24–25.
5. La norme C23 permet désormais d'initialiser un tableau de longueur variable à l'aide de l'initialisation « par défaut ». Voyez ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.10 Initialization, al. 4, p. 135 ainsi que la section « Une vraie initialisation à zéro » de ce billet.

```
13 static int i; /* 0 */
14 struct s s; /* 0, 0.0 et padding à zéro */
15 union u u; /* 0 et padding à zéro */
16 struct s t[2]; /* { 0, 0.0 } et padding à zéro, { 0, 0.0 } et
       padding à zéro */
```

Pour rappel, les *bytes* de *padding* d'une structure sont des *bytes* ajoutés entre ses membres ou à sa fin (mais jamais à son début[3]) en vue de respecter des contraintes d'alignement. Par exemple, il pourrait y avoir (et, typiquement, il y aura) des *bytes* de *padding* entre le champ `x` et le champ `y`. Il en va de même pour les unions, si ce n'est que les *bytes* de *padding* ne peuvent être qu'à sa fin[4].

Le souci, jusqu'à présent, c'est qu'en dehors des classes de stockage `static` et `thread_local`, il n'était pas facile d'obtenir le même résultat pour les structures et les unions. En effet, intuitivement, quatre solutions viennent en tête :

— Utiliser une initialisation incomplète ;
— Assigner une structure ou une union de classe de stockage `static` ou `thread_local` ;
— Utiliser la fonction `memset()` ;
— Copier une structure ou une union de classe de stockage `static` ou `thread_local` via la fonction `memcpy()`.

Cependant, seule la dernière solution est satisfaisante.

8.0.1. Initialisation incomplète

Dans le cas où tous les membres d'une structure ne se verraient pas assigner une valeur lors d'une initialisation, alors ils sont initialisés à zéro suivant les mêmes règles que pour les variables de classe de stockage `static` ou `thread_local`[5].

```
1  #include <stdio.h>
2
3  struct s {
4          int i;
5          double r;
6          void *p;
7  };
8
9  int
10 main(void) {
11         struct s s1 = { .i = 0 }; /* i = 0, r = 0.0 et p est un
               pointeur nul */
12         struct s s2 = { 0 }; /* Pareil */
13
14         printf("%f, %f\n", s1.r, s2.r);
```

```
15          return 0;
16  }
```

```
1  0.000000 0.000000
```

Le souci, c'est que cette règle ne vise *que les membres de la structure* et non les *bytes* de *padding*.

> *The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject ; **all subobjects** that are not initialized explicitly are subject to default initialization.*
>
> *ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.9 Initialization, al. 20, p. 136–137.*

C'est ennuyeux dans le cas où il est impératif que les *bytes* de *padding* soient garantis d'être à zéro.

8.0.2. Assignation d'une structure ou d'une union de classe de stockage `static` ou `thread_local`

On serait alors tenté d'assigner une structure ou une union de classe de stockage `static` ou `thread_local`, puisque leurs *bytes* de *padding* sont garantis d'être zéro. Cependant, la norme précise que lors d'une affectation, la valeur des *bytes* de *padding* d'une structure ou d'une union n'est pas spécifiée[6].

> *When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.*
>
> *ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.2.6 Representations of types, al. 6, p. 41.*

8.0.3. Utilisation de la fonction `memset()`

Une autre solution consiste à recourir à la fonction `memset()` pour mettre chaque *bytes* d'un objet à zéro. Cependant, un tel usage pose un problème : une « initialisation à zéro » ne signifie pas « tous les *bytes* à zéro » pour tous les types. En effet, un pointeur nul a une adresse spécifique qui n'est pas nécessairement zéro (le compilateur tenDRA , découvert via cet article , laisse par exemple la possibilité d'utiliser `0x55555555` comme adresse pour un pointeur nul). Dans le même sens, pour les nombres flottants, le zéro ne correspond pas nécessairement à « tous les *bytes* à zéro » (même si, suivant la norme IEEE 754 qui est massivement utilisée, il y a deux zéros : un positif et un négatif dont le premier a une telle représentation).

8.0.4. Copier une structure ou une union de classe de stockage `static` ou `thread_local` via la fonction `memcpy()`

Finalement, la dernière solution est de copier une structure ou une union de classe de stockage `static` ou `thread_local` via la fonction `memcpy()`, cette dernière effectuant une copie *bytes* par *bytes*, *padding* inclut[8]. Et, effectivement, cette solution rempli bien tous les critères.

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

struct s {
        int i;
        double r;
        void *p;
};

int
main(void) {
        static struct s s1; /* i = 0, r = 0.0, p est un pointeur
            nul et le padding est à zéro */
        struct s s2;

        memcpy(&s2, &s1, sizeof s1); /* i = 0, r = 0.0, p est un
            pointeur nul et le padding est à zéro */
        printf("%d, %f, %p\n", s2.i, s2.r, s2.p);
        return 0;
}
```

```
0, 0.000000, (nil)
```

8.0.5. La solution en C23

Pour en finir avec ceci, la norme C23 a introduit une initialisation « par défaut » qui peut être demandée en spécifiant `{}` comme initialisateur[7]. Ceci garantit une initialisation identique aux variables de classe de stockage `static` et `thread_local`.

```c
#include <stdio.h>

struct s {
        int i;
        double r;
        void *p;
};
```

```
8
9  int
10 main(void) {
11         struct s s = {}; /* i = 0, r = 0.0 et p est un pointeur
               nul + padding à zéro */
12
13         printf("%d, %f, %p\n", s.i, s.r, s.p);
14         return 0;
15 }
```

```
1  0, 0.000000, (nil)
```

Notez également que cette initialisation est autorisée pour les tableaux de longueur variable (*variable-length array* ou VLA). Il n'y avait aucun moyen d'initialiser de tels tableaux auparavant[9].

```
1  #include <stdio.h>
2
3  int
4  main(void) {
5          size_t n;
6
7          if (scanf("%zu", &n) == 1) {
8                  int tab[n] = {};
9
10                 for (size_t i = 0; i < n; i++) {
11                         printf("[%zu] = %d\n", i, tab[i]);
12                 }
13         }
14
15         return 0;
16 }
```

```
1  10
2  [0] = 0
3  [1] = 0
4  [2] = 0
5  [3] = 0
6  [4] = 0
7  [5] = 0
8  [6] = 0
9  [7] = 0
10 [8] = 0
11 [9] = 0
```

9. Typage des énumérations et de leurs membres

Jusqu'à présent, une énumération était décomposée en deux types :

— Le type de l'énumération, qui est un type entier capable de représenter toutes les valeurs des membres de l'énumération. Ce type pouvait être un type entier signé, non signé ou le type `char`[1], à la discrétion du compilateur ;
— Le type des membres de l'énumération, qui était nécessairement le type `int`[2].

Ces deux limitations ont été levées avec la norme C23 (bien qu'en vérité, la seconde l'était déjà en pratique[3]). Il est désormais possible de spécifier le type d'une énumération et de ses membres[4]. Également, dans le cas où aucun type n'est spécifié, le type des membres n'est plus limité au type `int`[6].

9.0.1. Type spécifié

```
1  #include <stdio.h>
2
3  #define INT_TYPE(expr) _Generic((expr), \
4          char: "char",                   \
5          unsigned char: "uchar",         \
6          signed char: "schar",           \
7          short: "short",                 \
8          unsigned short: "ushort",       \
9          int: "int",                     \
10         unsigned int: "uint",           \
11         long: "long",                   \
12         unsigned long: "ulong",         \
13         long long: "llong",             \
14         unsigned long long: "ullong",   \
15         default: "unknown")
16
17 enum chiffre : unsigned long {
18         UN = 1,
19         DEUX = 2,
20         TROIS = 3,
21         QUATRE = 4,
22         CINQ = 5,
```

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.10 Initialization, al. 11, p. 135.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.9 Initialization, al. 11, p. 135–136.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.2.1 Structure and union specifiers, al. 17 et 19, p. 104.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.2.1 Structure and union specifiers, al. 18–19, p. 104.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.9 Initialization, al. 20, p. 136–137.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.2.6 Representations of types, al. 6, p. 41.
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.10 Initialization, al. 11, p. 137.
8. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.26.2.1 The memcpy function, al. 1–3, p. 374.
9. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.10 Initialization, al. 4, p. 135.

```
        SIX = 6,
        SEPT = 7,
        HUIT = 8,
        NEUF = 9,
};

int
main(void) {
        enum chiffre chiffre;

        printf("chiffre est de type : %s\n", INT_TYPE(chiffre));
        printf("UN est de type: %s\n", INT_TYPE(UN));
        return 0;
}
```

```
chiffre est de type : ulong
UN est de type: ulong
```

Auparavant, le type de l'énumération aurait été choisi par le compilateur et ses membres auraient été de type `int`.

9.0.2. Type non spécifié

Dans le cas où un type n'est pas spécifié, le type de l'énumération sera, comme auparavant, choisi par le compilateur. Ce type doit être un type entier, signé ou non signé, à l'exception du type `bool` et des entiers de taille arbitraire[5]. Pour le type des membres, en revanche, les règles changent et viennent en fait valider et normaliser une pratique déjà en œuvre[3] : le type des membres sera le type `int` si ce dernier est capable de représenter toutes les valeurs des membres de l'énumération, sinon le type sera le même que celui de l'énumération[6].

Avec une subtilité toutefois : le type des membres n'est choisi qu'une fois la définition de l'énumération terminée (une fois le `}` atteint, autrement dit) [6]. Jusqu'à ce que ce point soit atteint, les règles suivantes s'appliquent et déterminent le type des membres [7].

- S'il n'y a pas d'autre membre le précédant et qu'aucune valeur n'est précisée, le membre sera de type `int` ;
- Si une valeur est précisée et qu'elle est représentable par le type `int`, le membre sera de type `int` ;
- Si une valeur est précisée et qu'elle n'est pas représentable par le type `int`, le membre sera du même type que la valeur assignée ;
- Le type du membre précédent avec pour valeur celle du membre précédent augmentée de un. Si cette addition induit un dépassement de capacité, alors un autre type entier capable de représenter la valeur augmentée de un sera choisi. Ce type sera signé si le type du membre précédent est signé et non signé sinon (faites attention que le caractère signé ou non signé est conservé, autrement dit, si vous dépassez le type `long` par exemple, le

type `unsigned long` ne sera pas choisi, car il est non signé). Dans le cas où aucun type ne peut représenter la valeur augmentée de un, la compilation échouera[8].

Ces règles permettent de fixer la valeur des membres sans être limité au type `int` et sans se soucier du type final.

```c
#include <stdio.h>
#include <limits.h>

#define INT_TYPE(expr) _Generic((expr), \
        char: "char",                   \
        unsigned char: "uchar",         \
        signed char: "schar",           \
        short: "short",                 \
        unsigned short: "ushort",       \
        int: "int",                     \
        unsigned int: "uint",           \
        long: "long",                   \
        unsigned long: "ulong",         \
        long long: "llong",             \
        unsigned long long: "ullong",   \
        default: "unknown")

enum test {
        INT = INT_MAX, /* int */
        BIGGER_THAN_INT, /* long */
        BIGGEST = ULLONG_MAX, /* unsigned long long */
};

int
main(void) {
        enum test test;
        printf("test est de type : %s\n", INT_TYPE(test));
        printf("INT est de type: %s\n", INT_TYPE(INT));
        printf("INT a pour valeur: %llu\n", (unsigned long
            long)INT);
        printf("BIGGER_THAN_INT est de type: %s\n",
            INT_TYPE(BIGGER_THAN_INT));
        printf("BIGGER_THAN_INT a pour valeur: %llu\n", (unsigned
            long long)BIGGER_THAN_INT);
        printf("BIGGEST est de type: %s\n", INT_TYPE(BIGGEST));
        printf("BIGGEST a pour valeur: %llu\n", (unsigned long
            long)BIGGEST);
        return 0;
}
```

```
test est de type : ulong
INT est de type: ulong
INT a pour valeur: 2147483647
BIGGER_THAN_INT est de type: ulong
BIGGER_THAN_INT a pour valeur: 2147483648
BIGGEST est de type: ulong
BIGGEST a pour valeur: 18446744073709551615
```

i Notez que le code a été compilé sur un Linux 64 *bits* où les types `unsigned long` et `unsigned long long` sont identiques. Si les types étaient différents, le type de l'énumération `test` devrait être `unsigned long long`.

10. Inférence de type avec auto et typeof

La norme C23 a ajouté deux mécanismes d'inférence de type basés sur des pratiques existantes des compilateurs : `auto` et `typeof`.

10.0.1. auto

La norme C23 modifie le sens du mot-clé `auto`, hérité du B et obsolète depuis fort longtemps. Il indique désormais, lors de la définition d'une variable, que son type doit être déduit de l'expression utilisée pour l'initialiser[1].

```c
auto n = 10;
auto p = &n;

/* Équivalent à */
int n = 10;
int *p = &n;
```

1. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 6.7.2.2 Enumeration specifiers, al. 4, p. 118.
2. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 6.7.2.2 Enumeration specifiers, al. 3, p. 118.
3. JeanHeyd Meneide, Shepherd (Shepherd's Oasis LLC),doc. N3029 , 19/07/2022, Improved Normal Enumerations, § 4.2. Using the Enumerators Midway in the Definition List.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, 6.7.2.2 Enumeration specifiers, al. 15, p. 109.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, 6.7.2.2 Enumeration specifiers, al. 12, p. 108.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, 6.7.2.2 Enumeration specifiers, al. 14, p. 109.
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, 6.7.2.2 Enumeration specifiers, al. 11, p. 108.
8. Notez que la norme autorise l'emploie de types entiers étendus. Dépasser le type `unsigned long long` ou `long long` peut donc conduire à l'emploie d'un de ces types (par exemple `__int128` du côté de GCC) et non à une erreur de compilation.

Ce mécanisme ressemble à celui intégré au langage C++ par la norme C++11, mais est en fait basé sur l'extension `__auto_type` du compilateur GCC et est plus limité. En effet, les restrictions suivantes s'appliquent [2] [3] :

— Il ne peut être utilisé que lors de la *définition* d'une variable ;
— La définition doit comprendre une initialisation ;
— L'identificateur en train d'être défini ne peut pas être utilisé au sein de l'initialisation ;
— Les définitions multiples sont interdites ;
— Le type inféré doit avoir été défini auparavant.

```c
auto a; /* Interdit, initialisation manquante */
auto b = 10, c = 20; /* Interdit, définition multiple */
auto d = d; /* Interdit, « d » ne peut pas être utilisé au sein de
    l'initialisation */

auto p = &(struct { int x; }) { .x = 10 }; /* Interdit, le type
    inféré n'est pas défini préalablement */
struct s;
auto q = &(struct s { int x; }) { .x = 10 }; /* Interdit, le type
    inféré n'est pas défini auparavant. */
struct t { int x; };
auto r = &(struct t) { .x = 10 }; /* Ok: (struct t*) */

auto x = 3.14; /* Ok: (double) */
auto y = x; /* Ok: (double) */
auto z = &y; /* Ok: (double*) */
```

À noter que le type inféré l'est *après* avoir appliqué les conversions usuelles, notamment la conversion d'un tableau en un pointeur sur son premier élément ou un identificateur de fonction en un pointeur de fonction. Dit autrement, il est notamment impossible d'inférer un type tableau[2].

```c
auto t[] = { 1, 2, 3, 4, 5 };
auto a = t; /* Équivalent à int *a = t; */
```

10.0.2. `typeof` et `typeof_unqual`

La norme C23 introduit les opérateurs `typeof` et `typeof_unqual` qui peuvent être utilisés à la place d'un type. Ces opérateurs produisent un nom de type sur base de leur opérande qui peut être soit une expression dont le type sera déduit, soit un nom de type[4].

Les opérateurs `typeof` et `typeof_unqual` sont identiques à la différence que `typeof_unqual` produit un type dépourvu de qualificateurs (`const`, `restrict`, `volatile` et `_Ato`

`mic`)[5].

```
1 int a = 10;
2 typeof(a) b = 20; /* Équivalent à int b = 20; */
```

Comme pour l'opérateur `sizeof`, si son opérande est une expression, elle n'est pas évaluée *sauf* s'il s'agit d'un tableau de taille variable[4].

```
1  #include <stdio.h>
2
3  int
4  main(void) {
5          int i = 0;
6          typeof(i++) j = 0; /* Équivalent à int j = 0; */
7          int *p = &i;
8          typeof(*p = 42) k = 0; /* Équivalent à int k = 0; */
9          printf("i = %d, j = %d, k = %d\n", i, j, k);
10
11         int n = 10;
12         typeof(int[n++]) m;
13         printf("n = %d\n", n);
14         return 0;
15 }
```

```
1 i = 0, j = 0, k = 0
2 n = 11
```

Comme vous le voyez, le type `int` a été déduit des expressions `i++` et `*p = 42`, mais aucune des ces expressions n'a été évaluée. Dit autrement, les opérations décrites ne sont pas effectuées, elles sont uniquement utilisées en vue de déduire un type. En revanche, le type `int[n]` étant un tableau de longueur variable, l'expression a cette fois-ci bien été évaluée.

11. Ajout des fonctions memccpy(), strdup() et strndup()

La norme C23 introduit trois nouvelles fonctions standards dans l'en-tête `<string.h>`. Il s'agit de trois fonctions en usage de longue date et provenant de la norme POSIX.

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.1 Storage-class specifiers, al. 4 et 14, p. 98 et 99.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.9 Type inference, al. 2, p. 133.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7 Declarations, al. 5 et 12, p. 96 et 97.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.2.5 Typeof specifiers, al. 4, p. 116.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.2.5 Typeof specifiers, al. 5, p. 116.

11.1. memccpy

La manipulation des chaînes de caractères en C n'est pas une sinécure et c'est un euphémisme de le dire. Une opération qui doit souvent être réalisée est la concaténation de chaînes, c'est-à-dire la fusion de plusieurs chaînes de caractères. La fonction standard qui est censée remplir ce rôle est la fonction `strcat()`, cependant, celle-ci souffre de deux défauts majeurs.

Le premier est qu'elle ne vérifie pas si la chaîne de destination à une taille suffisante pour se voir ajouter une autre chaîne à sa suite. Dans l'exemple ci-dessous, l'espace est suffisant, mais cela doit être vérifié et garanti par le programmeur en amont.

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

int
main(void) {
        char s[64];
        char *t[] = { "Première partie", ", ", "deuxième partie."
            };

        for (size_t i = 0; i < sizeof t / sizeof t[0]; i++) {
                (void)strcat(s, t[i]);
        }

        puts(s);
        return 0;
}
```

```
Première partie, deuxième partie
```

La seconde est qu'elle est assez inefficace, notamment parce que plusieurs concaténations supposent de parcourir la chaîne de destination jusqu'à sa fin à chaque concaténation. Dans l'exemple ci-dessus, le contenu de la chaîne `s` est parcouru trois fois : une fois avant d'ajouter `"Première partie` est parcourue, une fois avant d'ajouter `", "` et une fois avant d'ajouter `"deuxième partie"`. Il serait plus efficace de commencer les concaténations directement à la suite des autres, par exemple à l'aide d'un pointeur indiquant la fin d'une concaténation. Information que `strcat()` ne fournit pas, sa valeur de retour étant… son premier paramètre.

Différentes alternatives et solutions ont vu le jour au cours du temps, notamment la fonction standard `snprintf()` ou la fonction `strlcat()` du projet OpenBSD, mais la première amène une certaine lourdeur d'utilisation et la seconde ne résout que le premier problème. C'est ici que la fonction `memccpy()` fait son entrée.

```
void *memccpy(void * restrict s1, const void * restrict s2, int c,
    size_t n);
```

Cette dernière utilise quatre paramètres : une zone mémoire de destination `s1`, une zone mémoire source `s2`, une valeur d'arrêt `c` et un nombre de multiplets (*bytes*) `n`. Globalement, la fonction copie les données de la zone mémoire `s2`, multiplet (*byte*) par multiplet, vers la zone mémoire `s1` jusqu'à en avoir copié `n` multiplets **ou** jusqu'à ce qu'un multiplet ait la valeur `c` (ce multiplet est également copié dans `s1`). La fonction retourne un pointeur vers le multiplet *suivant* celui ayant la valeur `c` dans `s1` ou un pointeur nul si aucun multiplet n'a la valeur `c`[1].

Cette fonction remplit donc les deux critères précédemment décrits et nous permet d'adapter le code précédant comme suit.

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

int
main(void) {
        char s[64];
        char *t[] = { "Première partie", ", ", "deuxième partie."
            };
        char *p = s;

        for (size_t i = 0; i < sizeof t / sizeof t[0]; i++) {
                if ((p = memccpy(p, t[i], '\0', sizeof s - (p -
                    s))) == nullptr) {
                        s[sizeof s - 1] = '\0';
                        break;
                }

                p--;
        }

        puts(s);
        return 0;
}
```

```
Première partie, deuxième partie
```

Premier point : dans le cas où `memccpy()` retourne un pointeur nul, cela signifie qu'elle a terminé la copie sans rencontrer de caractère nul, ce qui signifie que la chaîne source est plus longue que la chaîne de destination et qu'**aucun caractère nul** ne se trouve dans la chaîne de destination. Il est donc nécessaire d'en ajouter un à sa fin, ce que nous faisons avec l'expression `s[sizeof s - 1] = '\0'`.

Second point : si le retour de `memccpy()` n'est pas un pointeur nul, il s'agit d'un pointeur vers le multiplet suivant le caractère nul rencontré et copié. Or, si nous voulons effectuer une autre concaténation, nous devons la commencer depuis le caractère nul et non après (sinon nous allons finir avec plusieurs chaînes de caractères et non une seule), il nous faut donc adapter le pointeur reçu, ce que nous faisons avec l'expression `p--`.

Dernier point : puisque nous ajoutons progressivement du contenu dans `s`, nous devons adapter la taille maximale à copier puisque l'espace disponible se réduit progressivement. Grâce au retour de `memccpy` cela peut se faire aisément à l'aide de la taille d'origine de `s` et d'un peu d'arithmétique des pointeurs ce qui donne, dans notre cas, l'expression `sizeof s - (p - s)`.

? Mais ? Pas si vite ! Si `memccpy()` retourne un pointeur vers le multiplet suivant le caractère nul copié, alors elle peut retourner un pointeur qui est au-delà de la zone mémoire, non ?

Excellente remarque ! C'est exact, si vous prenez le code suivant, le pointeur retourné par `memccpy()` pointe vers un multiplet qui suit le dernier multiplet du tableau `s`.

```c
#include <stdio.h>

int
main(void) {
        char s[2];
        char *p = memccpy(s, "a", '\0', sizeof s);
        printf("&s[0] = %p, &s[1] = %p, &s[2] = %p, p = %p\n",
            &s[0], &s[1], &s[2], p);
        return 0;
}
```

```
&s[0] = 0x7ffe671fc1e6, &s[1] = 0x7ffe671fc1e7, &s[2] =
    0x7ffe671fc1e8, p =  0x7ffe671fc1e8
```

Comme vous le voyez, l'adresse référencée par `p` est celle de `s[2]`. Cependant, cela ne pose pas de problème au regard de l'usage que nous en faisons. En effet, la norme nous garantit que l'adresse en question est valide et qu'elle peut être utilisée lors d'opération d'arithmétique des pointeurs.

> […] Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object
>
> *ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.5.6 Additive operators, al. 9, p. 84.*

> When two pointers are subtracted, both shall point to elements of the same array object, **or one past the last element of the array object**; the result is the difference of the subscripts of the two array elements.
>
> *ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.5.6 Additive operators, al. 10, p. 84.*

Ceci ne concerne *que* l'arithmétique des pointeurs, il est — en revanche — interdit d'accéder ou d'écrire à l'adresse suivant le tableau.

11.2. strdup et strndup

```
1 char *strdup(const char *s);
2 char *strndup(const char *s, size_t size);
```

La fonction `strdup()` retourne une copie de la chaîne `s` ou un pointeur nul si aucun espace mémoire n'a pu être allouée à l'aide de la fonction `malloc()`. La copie peut (ou plutôt, *doit*) être libérée à l'aide de la fonction `free()`[2].

La fonction `strndup()` fonctionne de la même manière si ce n'est qu'elle copie *au maximum* `size` caractères depuis `s`. La copie s'arrête lorsqu'elle rencontre un caractère nul (qui est copié) ou lorsqu'elle a atteint le maximum autorisé (auquel cas un caractère nul est ajouté à la fin de la copie)[3].

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int
5  main(void) {
6          char s[] = "Copiez-moi !";
7          char *c1 = strdup(s), *c2 = strndup(s, 6);
8
9          if (c1 != nullptr && c2 != nullptr) {
10                 puts(s);
11                 puts(c1);
12                 puts(c2);
13         }
14
15         return 0;
16 }
```

```
Copiez-moi !
Copiez-moi !
Copiez
```

12. Les fonctions à nombre variable d'arguments

Jusqu'à présent, une fonction à nombre variable d'arguments (dont la liste des paramètres se terminent par l'ellipse `...`) devait préciser au moins un paramètre avant l'ellipse. Dit autrement, il était exigé qu'au moins un paramètre soit fixe.

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

int sum(int len, ...) {
        va_list ap;
        va_start(ap, len);
        int res = 0;

        for (int i = 0; i < len; i++) {
                res += va_arg(ap, int);
        }

        va_end(ap);
        return res;
}

int
main(void) {
        printf("%d\n", sum(5, 1, 2, 3, 4, 5));
        return 0;
}
```

```
15
```

Si la présence d'un premier argument peut avoir du sens, par exemple comme ci-dessus pour indiquer le nombre d'arguments variables, ce n'est pas forcément le cas. Prenons par exemple le cas d'une fonction affichant un nombre variable de chaînes de caractères. Le nombre de chaînes n'est pas nécessaire, la fin pouvant être indiquée par un pointeur nul.

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.26.2.2 The memccpy function, al. 1–3, p. 374–375.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.26.2.6 The strdup function, al. 1–3, p. 375–376.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.26.2.7 The strndup function, al. 1–3, p. 376.

La norme C23 retire cette obligation, tout en conservant le second argument de la macrofonction `va_start()`, qui n'est plus utilisé[1] et ce, afin d'éviter aux anciens codes de ne plus compiler.

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

void
println(...) {
        va_list ap;
        va_start(ap);
        char const *s = va_arg(ap, char const *);

        while (s != NULL) {
                fputs(s, stdout);
                putchar(' ');
                s = va_arg(ap, char const *);
        }

        putchar('\n');
}

int
main(void) {
        println("Une", "multitude", "de", "chaînes", "de",
            "caractères", (char *)0);
        return 0;
}
```

```
Une multitude de chaînes de caractères
```

13. Prototypes obligatoires

Une conséquence de la modification de la macrofonction `va_start()` en vue de ne plus imposer au moins un paramètre aux fonctions à nombre variable d'arguments (voyez la section « Les fonctions à nombre variable d'arguments ») est l'abandon des déclarations de fonction sans liste de paramètres. Dit autrement, l'emploi des prototypes de fonctions est désormais obligatoire et ne pas préciser de paramètres revient à préciser que la fonction n'en reçoit aucun (comme si l'on avait écrit `void`)[1].

En effet, depuis ses débuts, le C traîne avec lui une ancienne manière de déclarer une fonction où la déclaration ne spécifie ni le nombre ni le type des paramètres[2].

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.16.1.4 The va_start macro, al. 1, p. 289.

```c
#include <stdio.h>

int multiplie();

int
main(void) {
        printf("2 * 2 = %d\n", multiplie(2, 2)); /* 2 * 2 = 4 */
        return 0;
}

int
multiplie(int a, int b) {
        return a * b;
}
```

Cette méthode conservait toutefois un intérêt en vue de disposer d'un pointeur de fonction « générique » (en dehors de son type de retour qui, lui, doit toujours être précisé).

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

struct input {
        void *from;
        int (*scanf)(); /* Pointeur de fonction « générique » */
};

int
main(void) {
        struct input in[] = {
                { stdin, &fscanf },
                { "1 2 3", &sscanf },
        };

        for (unsigned i = 0; i < sizeof in / sizeof in[0]; i++) {
                int a, b ,c;

                if (in[i].scanf(in[i].from, "%d %d %d", &a, &b,
                    &c) != 3) {
                        fprintf(stderr, "in[%u].scanf: %s\n", i,
                            strerror(errno));
                        exit(EXIT_FAILURE);
                }

                printf("%d, %d, %d\n", a, b, c);
        }
```

```
28
29          return 0;
30  }
```

```
1  1 2 3
2  1, 2, 3
3  1, 2, 3
```

Désormais, le pointeur `scanf` devra s'écrire `int (*scanf)(...)`, car en C23 `int (*scanf)()` est équivalent à `int (*scanf)(void)`.

14. Progression dans le support d'Unicode

La norme C11 avait introduit les types `char16_t` et `char32_t`[1] (respectivement synonymes des types `int_least16_t` et `int_least32_t`), les préfixes `u8`, `u` et `U` pour les chaînes de caractères littérales[2] ainsi que les fonctions `mbrtoc16()`, `c16rtomb()`, `mbrtoc32()` et `c32rtomb()`[3].

L'idée derrière était que les types `char16_t` et `char32_t` *pouvaient* être utilisés pour contenir des points de code UTF-16 ou UTF-32 et que les chaînes littérales préfixées avec `u` ou `U` *pouvaient* être encodées en UTF-16 ou UTF-32. De même, les fonctions de conversions introduites *pouvaient* convertir une chaîne de caractères (encodée suivant la locale courante) en UTF-16 ou en UTF-32 et inversement.

Cependant, *aucune garantie* n'était donnée par la norme sur ces points. Dans les faits, la seule garantie qui était donnée par la norme était qu'une chaîne préfixée par `u8` devait être encodée en UTF-8[2].

La norme C23 garantit désormais qu'une chaîne littérale respectivement préfixée par `u` ou `U` est encodée en UTF-16 ou en UTF-32[4]. Elle garantit également que les quatre fonctions de conversions ci-dessus convertissent une chaîne de caractères (encodée suivant la locale courante) en UTF-16 ou en UTF-32 et inversement[5].

La norme introduit également le type `char8_t`[6] (qui est un synonyme pour le type `unsigned char`) et qui est désormais le type des éléments d'une chaîne de caractères littérale préfixée par `u8`[4].

Enfin, les fonctions `mbrtoc8()` et `c8rtomb()` sont ajoutées. Ces dernières permettent de convertir une chaîne de caractères (encodée suivant la locale courante) en UTF-8 et inversement.

L'exemple ci-dessous tente de convertir une chaîne de caractères encodée selon la locale courante en UTF-32. La même logique peut être appliquée pour la conversion en UTF-8 ou en UTF-16.

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.6.3 Function declarators, al. 13, p. 128.
2. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 6.7.2.2 6.7.6.3 Function declarators (including prototypes), al. 14, p. 134.

```c
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <uchar.h>

int
main(void) {
	char buf[255];

	if (fgets(buf, sizeof buf, stdin) == NULL) {
		perror("fgets");
		exit(EXIT_FAILURE);
	}
	if (setlocale(LC_CTYPE, "") == NULL) {
		perror("setlocale");
		exit(EXIT_FAILURE);
	}

	char32_t utf32[sizeof buf];
	char *src = buf;
	char32_t *dst = utf32;
	mbstate_t state = {};
	size_t n;

	for (;;) {
		n = mbrtoc32(dst, src, MB_CUR_MAX, &state);

		if (n == 0) {
			break;
		}
		if (n > (size_t)-3) {
			perror("mbrtoc32");
			exit(EXIT_FAILURE);
		}
		if (n != (size_t)-3) {
			src += n;
		}

		dst++;
	}

	for (size_t i = 0; utf32[i] != U'\0'; i++) {
		printf("%04x ", utf32[i]);
	}

	fputc('\n', stdout);
	return 0;
```

```
}
```

```
Élégamment trouvé
00c9 006c 00e9 0067 0061 006d 006d 0065 006e 0074 0020 0074 0072
    006f 0075 0076 00e9 000a
□□
5e78 305b 000a
```

La fonction `mbrtoc32()` lit des *bytes* (au plus le maximum indiqué par son troisième paramètre, ici `MB_CUR_MAX`) depuis son second paramètre (ici `src`) et tente de convertir ces derniers en un point de code UTF-32. En cas de succès, elle écrit le `char32_t` dans la zone référencée par `dst`. Le quatrième argument est une variable permettant de stocker l'état actuel de la conversion, dans le cas de certains encodages à état comme l'ISO-2022-JP .

La fonction retourne :

— zéro, si elle lit une suite de *bytes* correspondant au caractère nul ;
— un nombre compris entre 1 et le nombre maximum de *bytes* pouvant être lu si elle a réussi à convertir cette suite de *bytes* en un point de code UTF-32 ;
— `(size_t)-3` si elle a écrit un point de code UTF-32 sans consommer de *bytes* depuis la chaîne d'origine (cela peut se produire avec certains encodages) ;
— `(size_t)-2` si elle a lu le maximum de *bytes* indiqué, mais que cette suite est insuffisante pour produire un point de code UTF-32 ;
— `(size_t)-1` si la suite de *bytes* lue est invalide et ne peut pas être convertie en un point de code UTF-32.

15. Vérification des dépassements de capacité des entiers

En C, un dépassement de capacité (*overflow* ou *underflow* en anglais) d'un entier *signé* est un comportement tantôt indéfini, tantôt dépendant de l'architecture, tantôt non spécifié. Un dépassement peut être la suite d'une opération[1] (addition, multiplication, etc.) ou d'une conversion[2] [3] [4] (implicite ou explicite). Il s'agit donc d'un comportement à éviter, le résultat étant très variable d'une machine à l'autre.

Pour les entiers non signés, la norme est plus précise en ce qui concerne les calculs et conversions entre entiers : la valeur boucle[5] [6]. Pour les autres conversions, les mêmes règles que pour les entiers signés s'appliquent. Cependant, même si le comportement est défini, il n'est pas forcément

1. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 7.28 Unicode utilities <uchar.h>, al. 2, p. 398.
2. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 6.4.5 String literals, al. 6, p. 71.
3. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 7.28.1 Restartable multibyte/wide character conversion functions, p. 398–401.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.4.5 String literals, al. 6, p. 67.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.30.1 Restartable multibyte/wide character conversion functions, al. 2, p. 407.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.30 Unicode utilities <uchar.h>, al. 3, p. 407.
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.30.1 Restartable multibyte/wide character conversion functions, p. 407–408.

souhaitable. En effet, imaginer un calcul de taille qui boucle (la taille sera donc nulle ou plus petite), ce n'est pas le résultat voulu.

Jusqu'à présent, il n'y avait pas de fonctions standards pour détecter les dépassements, tout devait être fait à la main. La norme C23 introduit trois macrofonctions : `ckd_add()`, `ckd_sub()` et `ckd_mul()`, définies dans le nouvel en-tête `<stdckdint.h>`[7].

```
bool ckd_add(type1 *result, type2 a, type3 b);
bool ckd_sub(type1 *result, type2 a, type3 b);
bool ckd_mul(type1 *result, type2 a, type3 b);
```

Ces trois macrofonctions effectuent respectivement la somme, la soustraction et la multiplication de `a` et `b` et stocke le résultat dans la variable référencée par `result`. L'opération est effectuée comme si les deux opérandes avaient un type entier signé de précision infinie. Le résultat est ensuite converti et stocké dans la variable référencée par `result`[8].

À noter que le type des variables `a` et `b` et celui pointé par `result` ne doivent pas nécessairement être identiques. En revanche, il doit s'agir de types entiers autres que `char` (`signed char` et `unsigned char` sont par contre permis), `bool`, qu'un entier de taille arbitraire (`_BitInt`) ou qu'un type énuméré[9].

Les trois fonctions retournent `true` en cas de dépassement et `false` dans le cas contraire[10] [11].

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

int
main(void) {
        int res;

        if (ckd_add(&res, INT_MAX - 1, 1)) {
                fprintf(stderr, "INT_MAX - 1 + 1: overflow\n");
                exit(EXIT_FAILURE);
        }

        printf("INT_MAX - 1 + 1 = %d\n", res);

        if (ckd_add(&res, INT_MAX, 1)) {
                fprintf(stderr, "INT_MAX + 1: overflow\n");
                exit(EXIT_FAILURE);
        }

        printf("INT_MAX + 1 = %d\n", res);
        return 0;
}
```

```
INT_MAX - 1 + 1 = 2147483647
INT_MAX + 1: overflow
```

16. Ajout de l'en-tête <stdbit.h>

Jusqu'à présent, le C ne fournissait aucune fonction permettant d'analyser la représentation binaire des entiers, tout devait être fait « à la main » à l'aide des opérateurs de manipulations des *bits* (`&`, `|`, `^`, `<` et `>`).

16.1. Boutisme

Le boutisme détermine l'ordre dans lequel les *bytes* sont organisés en mémoire. Il existe deux principaux boutismes : petit-boutiste (*little-endian*) et gros-boutiste (*big-endian*)[1]. Ils ne sont pas les seuls, mais il s'agit des deux agencements majoritaires.

Jusqu'à présent, le boutisme ne pouvait être déterminé qu'à l'aide de tests lors de l'exécution ou qu'à l'aide d'extensions proposées par les compilateurs. La norme C23 vient combler ce manque en introduisant la macroconstante `__STDC_ENDIAN_NATIVE__` dont la valeur est[2] :

— `__STDC_ENDIAN_LITTLE__` si la machine est petit-boutiste ;
— `__STDC_ENDIAN_BIG__` si la machine est gros-boutiste ;
— Une autre valeur si elle n'est ni petit-boutiste ni gros-boutiste.

16.2. Analyse de la représentation binaire

La norme introduit également plusieurs fonctions et macrofonctions permettant d'analyser la représentation binaire des types entiers *non signés*[3][4][5][6][7][8][9][10][11][12][13] :

— standards, à l'exception du type `bool` ;
— étendus ;

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.5 Expressions, al. 5, p. 71.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.3.1.3 Signed and unsigned integers, al. 3, p. 46.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.3.1.4 Real floating and integer, al. 1–2, p. 46.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.3.2.3 Pointers, al. 6, p. 49.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.2.5 Types, al. 11, p. 38.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.3.1.3 Signed and unsigned integers, al. 2, p. 46.
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.20.1 The ckd_ Checked Integer Operation Macros, al. 1, p. 311.
8. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.20.1 The ckd_ Checked Integer Operation Macros, al. 2, p. 311.
9. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.20.1 The ckd_ Checked Integer Operation Macros, al. 3, p. 311.
10. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.20.1 The ckd_ Checked Integer Operation Macros, al. 5, p. 311.
11. Notez que ces macrofonctions sont calquées sur les fonctions intégrées `__builtin_add_overflow()`, `__builtin_sub_overflow()` et `__builtin_mul_overflow()` de GCC.

— de taille arbitraire dont la grandeur (*width* en anglais) correspond à celle d'un type précédemment énuméré.

```c
#include <stdbit.h>
#include <stdio.h>

#define TEST(function, byte) printf("%s(%08b): %u\n", #function,
    (unsigned char)(byte), function((unsigned char)(byte)))
|
/*
 * « generic_return_type » et « generic_value type » correspondent
     au type de l'argument fourni.
 * Ici unsigned char.
 *
 * generic_return_type stdc_leading_zeros(generic_value_type
     value);
 * generic_return_type stdc_leading_ones(generic_value_type value);
 * generic_return_type stdc_trailing_zeros(generic_value_type
     value);
 * generic_return_type stdc_trailing_ones(generic_value_type
     value);
 * generic_return_type stdc_first_leading_zero(generic_value_type
     value);
 * generic_return_type stdc_first_leading_one(generic_value_type
     value);
 * generic_return_type stdc_first_trailing_zero(generic_value_type
     value);
 * generic_return_type stdc_first_trailing_one(generic_value_type
     value);
 * generic_return_type stdc_count_zeros(generic_value_type value);
 * generic_return_type stdc_count_ones(generic_value_type value);
 * bool stdc_has_single_bit(generic_value_type value);
 */

int
main(void) {
        TEST(stdc_leading_zeros, 0b0000'1111); /* Nombre de bits à
            0 depuis le bit de poids fort. */
        TEST(stdc_leading_ones, 0b1100'0000); /* Nombre de bits à
            1 depuis le bit de poids fort. */
        TEST(stdc_trailing_zeros, 0b0000'1000); /* Nombre de bits
            à 0 depuis le bit de poids faible. */
        TEST(stdc_trailing_ones, 0b0001'1111); /* Nombre de bits à
            1 depuis le bit de poids faible. */
        TEST(stdc_first_leading_zero, 0b1111'1110); /* Index+1 du
            premier bit valant 0 depuis le bit de poids fort,
            retourne zéro si aucun. */
```

```
30          TEST(stdc_first_leading_one, 0b0000'0000); /* Index+1 du
                premier bit valant 1 depuis le bit de poids fort,
                retourne zéro si aucun. */
31          TEST(stdc_first_trailing_zero, 0b1110'1111); /* Index+1 du
                premier bit valant 0 depuis le bit de poids faible,
                retourne zéro si aucun. */
32          TEST(stdc_first_trailing_one, 0b1100'0000); /* Index+1 du
                premier bit valant 1 depuis le bit de poids faible,
                retourne zéro si aucun. */
33          TEST(stdc_count_zeros, 0b0001'0011); /* Nombre de bits à
                0. */
34          TEST(stdc_count_ones, 0b1111'0011); /* Nombre de bits à 1.
                */
35          TEST(stdc_has_single_bit, 0b0001'0000); /* Retourne true
                si un seul et un seul bit est à 1. */
36          return 0;
37  }
```

```
1  stdc_leading_zeros(00001111): 4
2  stdc_leading_ones(11000000): 2
3  stdc_trailing_zeros(00001000): 3
4  stdc_trailing_ones(00011111): 5
5  stdc_first_leading_zero(11111110): 8
6  stdc_first_leading_one(00000000): 0
7  stdc_first_trailing_zero(11101111): 5
8  stdc_first_trailing_one(11000000): 7
9  stdc_count_zeros(00010011): 5
10 stdc_count_ones(11110011): 6
11 stdc_has_single_bit(00010000): 1
```

16.3. Calculs en rapport avec la représentation binaire

La norme définit également plusieurs fonctions et macrofonctions permettant d'effectuer des calculs en rapport avec la représentation binaire. Elles imposent les mêmes restrictions de type que les fonctions précédentes [14] [15] [16].

```
1  #include <stdbit.h>
2  #include <stdio.h>
3
4  #define TEST(function, value) printf("%s(%u): %u\n", #function,
       (value), function((value)))
5
6  /*
```

```
 * « generic_return_type » et « generic_value type » correspondent
     au type de l'argument fourni.
 * Ici unsigned int.
 *
 * generic_return_type stdc_bit_width(generic_value_type value);
 * generic_value_type stdc_bit_floor(generic_value_type value);
 * generic_value_type stdc_bit_ceil(generic_value_type value);
 */
int
main(void) {
        TEST(stdc_bit_width, 32769U); /* Nombre minimum de bits
            nécessaire pour représenter un nombre entier donné. */
        TEST(stdc_bit_floor, 32769U); /* La puissance de deux
            directement inférieure à un nombre donné.  */
        TEST(stdc_bit_ceil, 32769U); /* La puissance de deux
            directement supérieure à un nombre donné.  */
        return 0;
}
```

```
stdc_bit_width(32769): 16
stdc_bit_floor(32769): 32768
stdc_bit_ceil(32769): 65536
```

17. Nouvelles directives du préprocesseur et support des paramètres

La norme C23 introduit quatre nouvelles directives pour le préprocesseur : `#elifdef`, `#elifndef`, `#warning` et `#embed`. Elle ajoute également le support de paramètres pour les directives ainsi que trois opérateurs unaires pour les conditions : `__has_include`, `__has_embed` et `__has_c_attribute`. Enfin, elle ajoute la macrofonction `__VA_OPT__`.

1. Voyez ce chapitre du tutoriel C si vous ne connaissez pas ces deux représentations.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.2 Endian, al. 2–3, p. 302–303.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.3 Count Leading Zeros, p. 303.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.4 Count Leading Ones, p. 303.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.5 Count Trailing Zeros, p. 303–304.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.6 Count Trailing Ones, p. 304.
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.7 First Leading Zero, p. 304–305.
8. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.8 First Leading One, p. 305.
9. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.9 First Trailing Zero, p. 305–306.
10. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.10 First Trailing One, p. 306.
11. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.11 Count Zeros, p. 306–307.
12. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.12 Count Ones, p. 307.
13. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.13 Single-bit Check, p. 307–308.
14. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.14 Bit Width, p. 308.
15. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.15 Bit Floor, p. 308–309.
16. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.18.16 Bit Ceiling, p. 309.

17.1. #elifdef et #elifndef

Jusqu'à présent il existait deux raccourcis pour écrire des conditions : `#ifdef MACRO` et `#ifndef MACRO` respectivement pour `#if defined(MACRO)` et `#if !defined(MACRO)`. La norme C23 en introduit deux autres : `#elifdef MACRO` et `#elifndef MACRO`, respectivement pour `#elif defined(MACRO)` et `#elif !defined(MACRO)`[1].

17.2. #warning

Jusqu'ici, seule la directive `#error` permettait d'émettre un message d'erreur, mais en provoquant la fin de la compilation. La norme C23 ajoute la directive `#warning` qui permet d'émettre un avertissement sans pour autant stopper la compilation[2].

17.3. #embed

Cette nouvelle directive inclut le contenu d'un fichier (comme `#include`), mais sous la forme d'une liste d'initialisation comprenant une suite de constantes entières représentant chacune un *byte* du fichier [3] [4] [5] (un *byte* au sens du C, soit `CHAR_BIT` *bits*).

> *i* La norme autorise que la quantité de *bits* lue pour représenter chaque constante entière soit différente et fixée par un paramètre propre au compilateur, nous ne considérerons toutefois pas ce cas dans la suite de cette section.

Imaginons que nous souhaitions inclure le fichier suivant (au hasard… 🍊)[6].

```
            _.-|-/\-._
          \-'           '-.
        /    /\    /\    \/          _____                    ____
              _____ _____
      \/  <    .  >  ./.  \/        / ___ \                |  _ \ /
           ____|  __ \
  _   /  <         > /___\ |.      / /  / /___  ___  ____ | |_) |
      (___ | |  | |
.< \ /  <     /\    > ( #) |#)    / /  / / __ \/ _ \/ __ \|  _ <
    \___ \| |  | |
  | |    <       /\   -.   __\   / /__/ / /_/ /  __/ / / /| |_)
       |____) | |__| |
   \   <  <   V      > ).//_._(\  \_____/ .___/\___/_/ /_/
        |____/|_____/|_____/
  .)/\   <  <  .-     /  \_'_) )-..   /_/
      \  <   ./  /  > >       /._./
      /\   <  '-' >    >    /
        '-._ < v    >   _.-'
          / '-._____.-' \
```

```
14                  \/
```

```c
#include <stdio.h>

int
main(void) {
        unsigned char openbsd_art[] = {
#        embed "openbsd_art.txt"
        };

        printf("sizeof openbsd_art = %zu\n", sizeof openbsd_art);
        return 0;
}
```

```
sizeof openbsd_art = 761
```

Techniquement, nous l'avons dit, le fichier est lu *byte* par *byte*, pour produire une liste de constantes entières. Dans le cas où un *byte* a une taille de 8 *bits* (ce qui est le cas sur l'écrasante majorité des machines), notre code d'exemple ressemble à ceci après traitement par le préprocesseur.

Contenu masqué n°1

Dans le cas où la taille du fichier n'est pas divisible par celle d'un *byte* (à nouveau, au sens du C, soit `CHAR_BIT` *bits*), la compilation échouera[5].

À noter que cette liste ne doit pas nécessairement être utilisée pour initialiser un tableau d'`unsigned char` ni même un tableau, elle peut être utilisée dans n'importe quel contexte où une telle liste est valide.

17.3.1. Paramètres

La directive `#embed` supporte quatre paramètres qui permettent de modifier son comportement : `limit`, `prefix`, `suffix` et `if_empty`. Ces paramètres se placent à la fin de la directive.

17.3.1.1. limit

Le paramètre `limit` permet de fixer un maximum à la quantité de *byte* à lire depuis le fichier spécifié. Cette limite peut être plus grande que la taille du fichier, mais elle n'aura pas d'effet dans un tel cas (la lecture s'arrêtera une fois la fin du fichier atteinte). Elle peut également être nulle, auquel cas aucun *byte* ne sera lu depuis le fichier[7].

Ce paramètre est particulièrement utile lorsqu'il s'agit de lire depuis un fichier spécial qui n'a pas de fin, par exemple `/dev/urandom` sous Linux.

```c
#include <stdio.h>

int
main(void) {
        unsigned char tab[] = {
#        embed </dev/urandom> limit(10)
        };

        for (size_t i = 0; i < sizeof tab; i++) {
                printf("[%zu] = %d\n", i, tab[i]);
        }

        return 0;
```

```
[0] = 245
[1] = 203
[2] = 50
[3] = 208
[4] = 50
[5] = 190
[6] = 84
[7] = 57
[8] = 203
[9] = 84
```

17.3.1.2. prefix et suffix

Les paramètres `prefix` et `suffix` permettent, respectivement, d'insérer une suite de caractères avant, ou d'ajouter une suite de caractères après la liste qui sera produite. Aucun espace n'est implicitement ajouté entre le préfixe ou le suffixe et la liste produite. Si un espace est nécessaire, il doit être spécifié explicitement. Dans le cas où le fichier est vide, ces deux paramètres sont ignorés [8] [9].

phrase.txt

```
longue phrase.
```

```c
#include <stdio.h>

int
main(void) {
        const char s[] = {
#        embed "phrase.txt" prefix('U', 'n', 'e', ' ', ) suffix(, 0)
/*
 *        'U', 'n', 'e', ' ', 0x6c, 0x6f, 0x6e, 0x67, 0x75, 0x65,
 *        0x20, 0x70, 0x68, 0x72, 0x61, 0x73, 0x65, 0x2e, 0x0a, 0
 */
        };

        fputs(phrase, stdout);
        return 0;
}
```

```
Une longue phrase.
```

17.3.1.3. if_empty

Le paramètre `if_empty` permet de spécifier une liste alternative dans le cas où le fichier spécifié est vide. Si le fichier n'est pas vide, ce paramètre n'a aucun effet[10].

17.4. Support des paramètres

Bien que seule la directive `#embed` dispose de paramètres spécifiés par la norme, il s'agit d'un ajout général valable pour les autres directives, ce qui peut s'avérer intéressant pour de futures extensions proposées par les compilateurs. À noter que dans un tel cas, la norme encourage à préfixer les paramètres comme suit : `préfixe::paramètre`[11].

17.5. Opérateurs unaires

17.5.1. __has_include

Cet opérateur prend en argument un fichier à inclure (dans la même forme que la directive `#include`) et retourne `1` dans le cas où le fichier existe ou `0` sinon[12]. Cet opérateur permet de réaliser des tâches qui n'étaient jusqu'à présent faisables qu'à l'aide d'outils externes (les scripts `configure` par exemple).

```c
/* En-tête hypothétique. */
#if !__has_include(<minmax.h>)
#        define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

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

17.5.2. __has_embed

Cet opérateur fonctionne de manière similaire à `__has_include`, si ce n'est qu'il vérifie en plus si les paramètres fournis sont supportés et si le fichier n'est pas vide. La valeur produite est[13] [14] :

- `0` (représenté par la macroconstante `__STDC_EMBED_NOT_FOUND__`) si le fichier n'existe pas ou si un des paramètres fournit n'est pas supporté ;
- `1` (représenté par la macroconstante `__STDC_EMBED_FOUND__`) si le fichier existe, si tous les paramètres fournis sont supportés et si le fichier n'est pas vide ;
- `2` (représenté par la macroconstante `__STDC_EMBED_EMPTY__`) si le fichier existe, si tous les paramètres fournis sont supportés et si le fichier est vide.

```c
#include <stddef.h>

int
main(void) {
/*
 * Paramètre hypothétique fixant le nombre de bits lu pour
     composer chaque
 * constante entière de la liste ainsi que leur type.
 */
#if __has_embed("tab.bin") gcc::element_type(unsigned short)
        unsigned short words[] = {
#        embed("tab.bin") gcc::element_type(unsigned short)
        };
#elif __has_embed("tab.bin")
        unsigned char bytes[] = {
#        embed("tab.bin")
        };
        unsigned short words[sizeof bytes / sizeof(unsigned
            short)];

        for (size_t i = 0; i < sizeof bytes; i += sizeof
            *words) {
#        if __STDC_ENDIAN_NATIVE__ == __STDC_ENDIAN_LITTLE__
                words[i / sizeof *words] = bytes[i] |
                    bytes[i+1]<<8;
#        elif __STDC_ENDIAN_NATIVE__ == __STDC_ENDIAN_BIG__
                words[i / sizeof *words] = bytes[i]<<8 |
                    bytes[i+1];
#        else
#                error "Boutisme inconnu"
#        endif
#else
#error "Impossible d'intégrer le fichier tab.bin"
#endif
        }

        return 0;
}
```

17.5.3. __has_c_attribute

L'opérateur `__has_c_attribute` retourne `1` si l'attribut fourni en argument est supporté et `0` dans le cas contraire[15] (les attributs sont discutés dans une autre section de ce billet).

```
#if __has_c_attribute(fallthrough)
#        define FALLTHROUGH [[fallthrough]]
#elif __has_c_attribute(gcc::fallthrough)
#        define FALLTHROUGH [[gcc::fallthrough]]
#else
#        define FALLTHROUGH /* ... */
#endif
```

17.6. La macrofonciton __VA_OPT__

La macrofonction `__VA_OPT__` s'utilise dans le contexte d'une macrofonction à nombre variable d'arguments. Dans le cas où une macrofonction à nombre variable d'arguments reçoit au moins un argument variable, la macrofonction `__VA_OPT__` est remplacée par le texte fourni en argument, sinon elle est ignorée[16].

Cette macrofonction permet de corriger un problème qui existait depuis l'introduction des macrofonctions à nombre variable d'arguments : ces dernières devaient nécessairement recevoir au moins un argument variable. En effet, dans l'exemple ci-dessous, la macrofonction `DEBUG()` doit nécessairement recevoir un argument sans quoi le code produit est incorrect.

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

#define DEBUG(fmt, ...) debug(__FILE__, __func__, __LINE__, (fmt),
    __VA_ARGS__)

void
debug(const char *file, const char *func, int line, const char
    *fmt, ...) {
        fprintf(stderr, "%s: %s(): #%d: ", file, func, line);
        va_list ap;
        va_start(ap);
        vfprintf(stderr, fmt, ap);
        fputc('\n', stderr);
}

int
main(int argc, char **argv) {
        DEBUG("argc : %d, argv[0] = %s", argc, argv); /* Ok */
        DEBUG("Test"); /* Erreur */
        return 0;
}
```

```
main.c: In function 'main':
main.c:4:79: error: expected expression before ')' token
    4 | #define DEBUG(fmt, ...) debug(__FILE__, __func__,
        __LINE__, (fmt), __VA_ARGS__)
      |
                                ^
main.c:18:9: note: in expansion of macro 'DEBUG'
   18 |         DEBUG("Test"); /* Erreur */
      |
```

L'erreur est due au fait que le code produit finalement `debug("main.c", __func__, 18, ("Test"), );` qui est incorrect syntaxiquement. Cette situation peut être résolue en employant la macrofonction `__VA_OPT_` comme suit.

```
#define DEBUG(fmt, ...) debug(__FILE__, __func__, __LINE__, (fmt)
    __VA_OPT__(,) __VA_ARGS__)
```

Ainsi, la virgule ne sera présente que si la macrofonction `DEBUG()` reçoit au moins un argument variable.

18. Les attributs

Depuis longtemps, les compilateurs proposent des extensions sous la forme d'attributs. Ces attributs peuvent être appliqués à différents éléments, que ce soit des types, des identificateurs ou des instructions et permettent de fournir davantage d'information aux compilateurs, par exemple pour ouvrir la voie à certaines optimisations.

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.6 Diagnostic directives, al. 1, p. 185.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.1 Conditional inclusion, al. 14, p. 166–167.
3. La norme ne parle pas de *byte* pour décrire cette directive, la taille d'un *byte* pouvant ne pas être identique pour un programme C et pour le système de fichier de la machine cible. Toutefois, dans l'écrasante majorité des cas, la taille d'un *byte* est identique pour les deux. Voyez la section « *Support Level 0, Part II : Preprocessor Parameters* » de cet article .
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.3.1 #embed preprocessing directive, al. 10, p. 171.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.3.1 #embed preprocessing directive, al. 6, p. 170.
6. Source : https://github.com/rbaylon/bsdsplash .
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.3.2 limit parameter, al. 3–4, p. 173.
8. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.3.4 prefix parameter, al. 2–3, p. 175.
9. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.3.3 suffix parameter, al. 2–3, p. 175.
10. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.3.5 if_empty parameter, al. 2, p. 176.
11. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10 Preprocessing directives, al. 4–5, p. 164.
12. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.1 Conditional inclusion, al. 6, p. 165.
13. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.1 Conditional inclusion, al. 7, p. 165–166.
14. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.9.1 Mandatory macros, al. 1, p. 186.
15. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.1 Conditional inclusion, al. 9, p. 166.
16. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.10.4.1 Argument substitution, al. 3, p. 178.

Un attribut assez commun est par exemple l'attribut « *packed* » qui permet d'éviter la présence de *bytes* de *padding* au sein d'une structure.

Jusqu'à présent, chaque compilateur utilisait sa propre syntaxe et le concept n'existait pas au sein de la norme C. La norme C23 introduit désormais le concept et définit sept attributs standards : `deprecated`, `fallthrough`, `maybe_unused`, `nodiscard`, `noreturn`, `unsequenced` et `reproducible`[1].

Un attribut est indiqué en insérant son nom entre double crochets `[[attribut]]`. Ces doubles crochets peuvent comporter un nom d'attribut ou plusieurs, séparés par des virgules `[[premier_attribut, deuxième_attribut]]`[2]. Pour les autres attributs, la norme recommande à chaque compilateur d'utiliser un préfixe propre sous la forme `préfixe::` placé juste avant le nom de l'attribut `[[préfixe::attribut_non_standard]]`[3].

18.1. deprecated

L'attribut `deprecated` peut être appliqué à :

— Une structure, une union, une énumération ou à un de leur membre ;
— Une définition de type (`typedef`) ;
— Une déclaration de variable ou de fonction[4].

Cet attribut permet d'indiquer à un utilisateur que l'utilisation de certains éléments est découragée, typiquement parce qu'ils vont bientôt disparaître. On peut penser à une bibliothèque qui revoit son API et encourage ses utilisateurs à ne plus utiliser certaines fonctions[5].

```c
#include <stdio.h>

struct [[deprecated("Ce type sera supprimé prochainement")]] coord {
        int x;
        int y;
};

int
main(void) {
        struct coord c = { 10, 20 };
        printf("%d, %d\n", c.x, c.y);
        return 0;
}
```

```
main.c: In function 'main':
main.c:10:16: warning: 'coord' is deprecated: Ce type sera supprimé prochainement [-Wdeprecated-declarations]
   10 |         struct coord c = { 10, 20 };
      |                ^~~~~
```

```
main.c:3:63: note: declared here
    3 | struct [[deprecated("Ce type sera supprimé
      |                     prochainement")]] coord {
      |
```

Notez que cet attribut accepte une chaîne de caractères en argument afin d'afficher un message lors de la compilation.

18.2. fallthrough

L'attribut `fallthrough` permet de supprimer les avertissements du compilateur dans le cas où, au sein d'une instruction `switch`, une ou plusieurs étiquettes (`case` ou `default`) sont accessibles depuis une ou plusieurs autres étiquettes[6].

Dans l'exemple ci-dessous, `case 1` et `case 2` sont accessibles depuis `case 3` et `case 1` est accessible depuis `case 2`. C'est un comportement délibérément voulu dans le cas de notre exemple, mais le compilateur émettra un avertissement, ce comportement étant rarement voulu.

```c
#include <stdio.h>


int
main(void) {
        unsigned nb = 0;

        while (nb == 0 || nb > 3) {
                printf(
                    "Combien de fois souhaitez-vous répéter l'affichage (entre
                    );
                scanf("%u", &nb);
        }

        switch (nb) {
        case 3:
                printf("Cette phrase est répétée une à trois fois.
                    \n");
        case 2:
                printf("Cette phrase est répétée une à trois fois.
                    \n");
        case 1:
                printf("Cette phrase est répétée une à trois fois.
                    \n");
        }

        return 0;
}
```

```
main.c: In function 'main':
main.c:15:17: warning: this statement may fall through
    [-Wimplicit-fallthrough=]
   15 |                 printf("Cette phrase est répétée une à
       trois fois.\n");
      |
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.c:16:9: note: here
   16 |         case 2:
      |         ^~~~
main.c:17:17: warning: this statement may fall through
    [-Wimplicit-fallthrough=]
   17 |                 printf("Cette phrase est répétée une à
       trois fois.\n");
      |
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.c:18:9: note: here
   18 |         case 1:
      |
```

L'attribut `fallthrough` peut être placé juste avant une étiquette (`case` ou `default`) accessible depuis une ou plusieurs autres étiquettes. Pour ce faire, on le déclare seul juste avant la ou les étiquettes accessibles[7].

```c
switch (nb) {
case 3:
        printf("Cette phrase est répétée une à trois fois.\n");
        [[fallthrough]];
case 2:
        printf("Cette phrase est répétée une à trois fois.\n");
        [[fallthrough]];
case 1:
        printf("Cette phrase est répétée une à trois fois.\n");
}
```

18.3. maybe_unused

L'attribut `maybe_unused` peut être appliqué à :

- Une structure, une union, une énumération ou à un de leur membre ;
- Une définition de type (`typedef`) ;
- Une déclaration de variable ou de fonction ;
- une étiquette (pour l'instruction `goto`) [8].

Cet attribut permet d'indiquer qu'un élément est inutilisé intentionnellement[9]. Un exemple typique est la non-utilisation d'un paramètre au sein d'une fonction.

Dans l'exemple ci-dessous, la fonction `nothing()` reçoit un paramètre, mais ne l'utilise pas, ce qui génère un avertissement de la part du compilateur.

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

int
nothing(void *unused) {
        return 2;
}

int
main(void) {
        thrd_t td;

        if (thrd_create(&td, &nothing, nullptr) == thrd_success) {
                int res = 0;

                if (thrd_join(td, &res) == thrd_success) {
                        printf("res = %d\n", res);
                }
        }

        return 0;
}
```

```
main.c: In function 'nothing':
main.c:6:15: warning: unused parameter 'unused'
    [-Wunused-parameter]
    6 | nothing(void *unused) {
      |
```

Afin d'éviter ce message, il est possible de marquer le paramètre `unused` avec l'attribut `maybe_unused`.

```c
int
nothing([[maybe_unused]] void *unused) {
        return 2;
}
```

18.4. nodiscard

L'attribut `nodiscard` peut être appliqué à une structure, une union, une énumération ou une fonction[10].

Cet attribut s'applique dans le cas d'un appel de fonction dont la valeur de retour est ignorée. Si la fonction ou son type de retour a été marqué avec l'attribut `nodiscard`, le compilateur produira un avertissement. Comme pour l'attribut `deprecated`, une chaîne de caractères peut être fournie en argument[11].

```
1  #include <stdio.h>
2
3  [[nodiscard("Le retour de scanf doit être vérifié !")]] extern int
       scanf(const char * restrict fmt, ...);
4
5  int
6  main(void) {
7          int age;
8
9          printf("Quel âge avez-vous ? ");
10         scanf("%d", &age);
11         printf("Vous avez %d an(s)\n", age);
12         return 0;
13 }
```

```
1  main.c: In function 'main':
2  main.c:10:9: warning: ignoring return value of 'scanf', declared
       with attribute 'nodiscard': "Le retour de scanf doit être
       vérifié !" [-Wunused-result]
3     10 |         scanf("%d", &age);
4        |         ^~~~~~~~~~~~~~~~~
```

18.5. noreturn

L'attribut `noreturn` peut être appliqué à une fonction[12]. Une fonction marquée avec l'attribut `noreturn` est supposée ne pas rendre la main à la fonction appelante (par exemple `abort()` ou `exit()`). Dans le cas où la fonction est susceptible de rendre la main à la fonction qui l'a appelée, le compilateur produira un avertissement[13].

```
1  #include <stdlib.h>
2
3  [[noreturn]] void
4  fonction(int i) {
```

```
5           if (i > 0) {
6                   abort();
7           }
8   }
9
10  int
11  main(void) {
12          return 0;
13  }
```

```
1   main.c: In function 'fonction':
2   main.c:8:1: warning: 'noreturn' function does return
3       8 | }
4         | ^
```

18.6. La machine abstraite

Avant de parler des attributs `reproductible` et `unsequence`[14], il est bon de rappeler ou de présenter le concept de *machine abstraite*. Afin de garantir que l'exécution d'un programme soit identique quel que soit le compilateur utilisé ou la machine sur laquelle il est exécuté, la norme décrit, de manière abstraite, comment un programme est supposé se comporter notamment en imposant un ordre dans l'évaluation des expressions et l'application des effets de bords (par exemple la modification d'une variable ou l'écriture dans un fichier)[15].

C'est cet ordre imposé qui garantit par exemple que la suite d'instructions `a = 10; a++;` conduit bien à ce que `a` ait pour valeur `11` et non `10` ou autre chose. Cet ordre repose sur des *points de séquences* qui fixent des instants où les évaluations et effets de bords doivent avoir eu lieu[16]. Dans notre exemple, il y a un point de séquence entre `a = 10;` et `a++;` ce qui nous garantit que `a` vaut bien 10 *avant* d'être incrémenté. On dit que l'évaluation de `a = 10` et *séquencée* avant `a++`.

Cependant, comme indiqué, il s'agit d'une description abstraite qui ne correspond pas au fonctionnement des machines sur lesquelles les programmes sont effectivement exécutés. Aussi, la norme n'impose pas que le programme s'exécute comme décrit, mais qu'il se comporte « comme si » il était exécuté sur cette machine abstraite[17] (la norme parle de « comportement observable »). C'est cette nuance qui ouvre la voie aux optimisations .

En effet, en suivant cette logique, un compilateur peut parfaitement remplacer `a = 10; a++;` par `a = 11;`, car le comportement observable est identique : à la fin `a` vaut bien 11.

Ceci étant posé, nous pouvons passer aux deux derniers attributs ajoutés par la norme C23. 🍏

18.7. unsequenced

L'attribut `unsequenced` ne peut être appliqué qu'à une fonction[18]. Cet attribut indique que des appels successifs à cette fonction avec les mêmes arguments produiront le même résultat indépendamment d'éventuels effets de bords produits entre ces appels. Sa valeur de retour n'étant pas affectée par des effets de bords, son appel peut être « non séquencé » (comprendre : il n'a pas besoin d'être séquencé comme spécifié par la norme)[19].

Dans le code ci-dessous, nous marquons la fonction `sqrt()` comme `unsequenced`. Cette information peut permettre au compilateur de ne l'appeler qu'une seule fois et de réutiliser cette valeur durant les itérations de la boucle.

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

int
main(void) {
        extern double sqrt(double) [[unsequenced]];
        double total = 0.;

        for (int i = 0; i < 10; i++) {
                total += sqrt(5.) + i;
        }

        printf("%f\n", total);
        return 0;
}
```

```
67.360680
```

Dit autrement, la boucle peut être réécrite comme suit par le compilateur.

```
double square = sqrt(5.);

for (int i = 0; i < 10; i++) {
        total += square + i;
}
```

18.8. reproductible

L'attribut `reproductible` ne peut être appliqué qu'à une fonction[20]. Cet attribut est similaire à l'attribut `unsequenced`, mais en fournissant un peu moins de garanties. Comme pour `unsequenced`, il indique que des appels successifs à une fonction produiront le même résultat si elle

reçoit les mêmes arguments *et si le comportement observable par la fonction n'a pas été modifié entre-temps*[21].

Deux exemples de fonctions standards pouvant être marquées comme `reproductible` sont `strlen()` et `strcmp()`. Elles ne peuvent pas être marquées comme `unsequenced` car elles lisent des zones mémoires (des chaînes de caractères) qui sont susceptibles d'être modifiées entre deux appels.

Dans l'exemple ci-dessous, nous marquons la fonction `strlen()` comme `reproductible`. Comme pour `sqrt()` dans l'exemple précédent, cette information peut permettre au compilateur de ne l'appeler qu'une seule fois et de réutiliser cette valeur durant les itérations de la boucle, à condition que le compilateur puisse garantir que la chaîne n'est pas modifiée durant les différentes itérations.

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

int
main(void) {
        char buf[255];

        if (fgets(buf, sizeof buf, stdin) != NULL) {
                extern size_t strlen(const char *)
                    [[reproductible]];

                for (size_t i = 0; i < strlen(buf); i++) {
                        if (buf[i] == '\n') {
                                break;
                        }

                        printf("[%02zu] = '%c'\n", i, buf[i]);
                }
        }

        return 0;
}
```

```
Test
[00] = 'T'
[01] = 'e'
[02] = 's'
[03] = 't'
```

Dit autrement, la boucle peut être réécrite comme suit par le compilateur.

```c
size_t len = strlen(buf);

for (size_t i = 0; i < len; i++) {
        if (buf[i] == '\n') {
                break;
        }

        printf("[%02zu] = '%c'\n", i, buf[i]);
}
```

19. Nouvel indicateur de taille pour printf et scanf

Depuis la norme C99, plusieurs nouvelles définitions de type (`typedef`) ont été introduites pour désigner les types entiers via l'en-tête `<stdint.h>`. Plus précisément, trois formes distinctes ont été ajoutées :

— `intXX_t` et `uintXX_t` [1] ;
— `int_leastXX_t` et `uint_leastXX_t` [2] ;
— `int_fastXX_t` et `uint_fastXX_t` [3].

où `XX` représente la grandeur du type entier en *bits* pour la pemière forme (la grandeur étant le nombre de *bits* de valeur ainsi qu'un éventuel *bit* de signe) et la taille du type entier en *bits*

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.1 General, al. 2, p. 141.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.1 General, al. 1, p. 141.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.1 General, al. 5, p. 142.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.4 The deprecated attribute, al. 1, p. 144.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.4 The deprecated attribute, al. 6, p. 144.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.5 The fallthrough attribute, al. 3, p. 145.
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.5 The fallthrough attribute, al. 1, p. 145.
8. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.3 The maybe_unused attribute, al. 1, p. 143.
9. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.3 The maybe_unused attribute, al. 4, p. 143.
10. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.2 The nodiscard attribute, al. 1, p. 142.
11. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.2 The nodiscard attribute, al. 4–5, p. 142.
12. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.6 The noreturn and _Noreturn attributes, al. 2, p. 146.
13. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.6 The noreturn and _Noreturn attributes, al. 6, p. 146.
14. Notez que ces deux attributs sont empruntés à GCC, l'attribut `unsequenced` correspond à l'attribut `const` et l'attribut `reproductible` à l'attribut `pure`.
15. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 5.1.2.3 Program execution, al. 3, p. 13.
16. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, Annex C Sequence points, p. 500.
17. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 5.1.2.3 Program execution, al. 6, p. 13.
18. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.7.2 The unsequenced type attribute, al. 1, p. 148.
19. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.7.2 The unsequenced type attribute, al. 6, p. 149.
20. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.7.1 The reproducible type attribute, al. 1, p. 148.
21. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.7.12.7.1 The reproducible type attribute, al. 3, p. 148.

(pouvant donc inclure des *bits* de *padding*) pour les deux autres.

Seules les deux dernières formes avec un nombre de *bits* de 8, 16, 32 et 64 sont rendues obligatoires par la norme, les autres sont facultatives.

Toutefois, la norme C23 impose désormais que si une définition de type `intXX_t` ou `uintXX_t` existe, alors les définitions de type `int_leastXX_t` et `uint_leastXX_t` correspondante doivent être identiques (par exemple, si `int32_t` existe, alors `int_least32_t` référence le même type)[4].

Jusqu'à présent, l'usage de ces types avec les fonctions des familles `printf()` et `scanf()` était assez fastidieux. En effet, il est nécessaire d'employer des macroconstantes définies dans l'en-tête `<inttypes.h>` qui rendent l'ensemble assez peu lisible[5].

```c
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>

int
main(void) {
        int32_t a;
        uint_least64_t b;
        int_fast16_t c;

        printf("Veuillez entrer trois nombres entiers : ");

        if (scanf("%" SCNd32 "%" SCNuLEAST64 "%" SCNdFAST16, &a,
            &b, &c) == 3) {
                printf("a = %" PRId32 "\n", a);
                printf("b = %" PRIuLEAST64 "\n", b);
                printf("c = %" PRIdFAST16 "\n", c);
        }

        return 0;
}
```

```
Veuillez entrer trois nombres entiers : 1 2 3
a = 1
a = 2
a = 3
```

Afin de simplifier leur utilisation, la norme C23 introduit deux indicateurs de taille : `w` (pour les formes `intxx_t`, `uintXX_t`, `int_leastXX_t` et `uint_leastXX_t`) et `wf` (pour les formes `int_fastXX_t` et `uint_fastXX_t`). Ceux-ci sont suivis d'une taille en *bits* puis d'un indicateur

de conversion entier (`b`, `d`, `i`, `o`, `u`, `x` ou `X`)[6][7].

```c
#include <stdint.h>
#include <stdio.h>

int
main(void) {
        int32_t a;
        uint_least64_t b;
        int_fast16_t c;

        printf("Veuillez entrer trois nombres entiers : ");

        if (scanf("%w32d %w64u %wf16d", &a, &b, &c) == 3) {
                printf("a = %w32d\n", a);
                printf("b = %w64u\n", b);
                printf("c = %wf16d\n", c);
        }

        return 0;
}
```

```
1
2
3
```

20. Classe de stockage pour les littéraux agrégats

Les littéraux agrégats permettent de définir des objets[1] sans nécessité d'utiliser des variables. Leur syntaxe a la forme `(type) { initialisation }`.

```c
#include <stdio.h>

struct exemple {
```

1. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.22.1.1 Exact-width integer types, al. 1–3, p. 315.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.22.1.2 Minimum-width integer types, al. 1–4, p. 315–316.
3. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.22.1.2 Fastest minimum-width integer types, al. 1–3, p. 316.
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.22.1.2 Minimum-width integer types, al. 3, p. 315.
5. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.8.1 Macros for format specifiers, al. 1–7, p. 22.
6. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.23.6.1 The fprintf function, al. 7, p. 330.
7. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.23.6.2 The fscanf function, al. 11, p. 337–338.

```
4           int x;
5   };
6
7   double *p = (double[]){ 1., 2., 3. };
8
9   int
10  main(void) {
11          printf("%d\n", (int){ 10 });
12          printf("%d\n", (int[]){ 1, 2, 3 }[1]);
13          printf("%d\n", (struct exemple){ .x = 5 }.x);
14          printf("%p\n", (void *)&(int){ 20 });
15          printf("%f\n", p[2]);
16  }
```

```
1  10
2  2
3  5
4  0x7ffd1fb92fd8
5  3.000000
```

La classe de stockage de ces objets dépend de leur contexte d'utilisation. S'ils sont utilisés au sein d'un bloc, ils sont de classe de stockage automatique, s'ils sont utilisés en dehors de tout bloc, leur classe de stockage est statique. Dans notre exemple, seul l'objet `(double){ 1., 2., 3. }` et le pointeur `p` seront de classe de stockage statique.

Cependant, tout comme il est possible de modifier la classe de stockage d'une variable, il serait intéressant de pouvoir également modifier la classe de stockage des littéraux agrégats.

```
1   struct illustration {
2           void *p;
3           int y;
4           int z;
5   };
6
7   struct exemple {
8           int x;
9           struct illustration *i;
10  };
11
12  int
13  main(void) {
14          static struct illustration illustration = { .p = (void
                *)1, .y = 20, .z = 30 };
15          static struct exemple exemple = { .x = 10, i =
                &illustration };
16          return 0;
```

```
17 }
```

Dans l'exemple ci-dessus, nous sommes contraints de commencer par définir une variable `illustration` pour pouvoir ensuite initialiser `exemple` avec son adresse. La norme C23 vient simplifier ce genre de cas en permettant de modifier la classe de stockage d'un littéral agrégat à l'aide d'un des spécificateurs `static`, `thread_local`, `constexpr` ou `register`[2].

```
1  struct illustration {
2          void *p;
3          int y;
4          int z;
5  };
6
7  struct exemple {
8          int x;
9          struct illustration *i;
10 };
11
12 int
13 main(void) {
14         static struct exemple exemple = {
15                 .x = 10,
16                 i = &(static struct illustration){ .p = (void *)1,
                       .y = 20, .z = 30 },
17         };
18         return 0;
19 }
```

21. Changement de définition pour intmax_t et uintmax_t

La norme C99 avait introduit les types `intmax_t` et `uintmax_t`. Ces types devaient désignés le type entier avec la capacité la plus élevée, types entiers non standards inclus[1]. Certaines fonctions utilisant ces types ont été introduites au même moment, comme `imaxabs()`[2].

Seulement voilà, l'introduction de ces types a fini par poser un problème lié à l' ABI du C. Sans entrer dans les détails[3], le fait que ces types varient en fonction des plus grands entiers disponibles sur une machine pose des problèmes lors de l'utilisation de fonctions provenant de bibliothèques tierces (comprendre : qui n'ont pas été compilées sur la même machine) et utilisant ce type.

En attendant de fournir une solution globale à ce problème, la norme C23 a modifié la définition des types `intmax_t` et `uintmax_t` afin qu'ils puissent se limiter à désigner le type `long`

1. pour rappel, dans le contexte du C, un objet est une zone de stockage de données, voyez ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 3.15 object, al. 1–2, p. 6.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 6.5.2.5 Compound literals, al. 5, p. 78.

`long`[4].

Conclusion

Si cette nouvelle version de la norme C n'apporte aucun changement majeur au langage, elle amène toutefois son lot d'améliorations et de nouvelles fonctionnalités intéressantes. Il n'y a plus qu'à attendre un support complet des compilateurs et des implémentations de la bibliothèque standard.

Autres articles sur le sujet

— (fr) Nouveautés du langage C dans sa prochaine version C23 ;
— (en) C23 is Finished : Here is What is on the Menu ;
— (en) Implementing #embed for C and C++ ;

Contenu masqué

Contenu masqué n°1

```c
#include <stdio.h>

int
main(void) {
        unsigned char openbsd_art[] = {
                0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                    0x20, 0x20, 0x20, 0x20,
                0x5f, 0x2e, 0x2d, 0x7c, 0x2d, 0x2f, 0x5c, 0x2d,
                    0x2e, 0x5f, 0x20, 0x20,
                0x20, 0x20, 0x20, 0x0a, 0x20, 0x20, 0x20, 0x20,
                    0x20, 0x20, 0x20, 0x20,
                0x20, 0x5c, 0x2d, 0x27, 0x20, 0x20, 0x20, 0x20,
                    0x20, 0x20, 0x20, 0x20,
                0x20, 0x20, 0x27, 0x2d, 0x2e, 0x20, 0x20, 0x20,
                    0x20, 0x0a, 0x20, 0x20,
                0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x2f, 0x20,
                    0x20, 0x20, 0x20, 0x2f,
                0x5c, 0x20, 0x20, 0x20, 0x20, 0x2f, 0x5c, 0x20,
                    0x20, 0x20, 0x20, 0x5c,
```

1. ISO/IEC 9899:201x, doc. N1570 , 12/04/2011, § 7.20.1.5 Greatest-width integer types, al. 1, p. 291.
2. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.8.2.1 The imaxabs function, al. 1–3, p. 223.
3. Voyez cet article pour les détails complets : https://thephd.dev/to-save-c-we-must-save-abi-fixing-c-function-abi
4. ISO/IEC 9899:2023, doc. N3096 , 01/04/2023, § 7.22.1.5 Greatest-width integer types, al. 1, p. 316.

Contenu masqué

```
13                  0x2f, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x5f,
14                  0x5f, 0x5f, 0x5f, 0x5f, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
15                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x5f, 0x5f, 0x5f,
16                  0x5f, 0x20, 0x20, 0x20, 0x5f, 0x5f, 0x5f, 0x5f,
                        0x5f, 0x20, 0x5f, 0x5f,
17                  0x5f, 0x5f, 0x5f, 0x20, 0x20, 0x0a, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
18                  0x5c, 0x2f, 0x20, 0x20, 0x3c, 0x20, 0x20, 0x20,
                        0x20, 0x2e, 0x20, 0x20,
19                  0x3e, 0x20, 0x20, 0x2e, 0x2f, 0x2e, 0x20, 0x20,
                        0x5c, 0x2f, 0x20, 0x20,
20                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x2f, 0x20,
                        0x5f, 0x5f, 0x5f, 0x20,
21                  0x5c, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
22                  0x20, 0x20, 0x20, 0x20, 0x7c, 0x20, 0x20, 0x5f,
                        0x20, 0x5c, 0x20, 0x2f,
23                  0x20, 0x5f, 0x5f, 0x5f, 0x5f, 0x7c, 0x20, 0x20,
                        0x5f, 0x5f, 0x20, 0x5c,
24                  0x20, 0x0a, 0x20, 0x20, 0x5f, 0x20, 0x20, 0x20,
                        0x2f, 0x20, 0x20, 0x3c,
25                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x3e, 0x20, 0x2f,
26                  0x5f, 0x5f, 0x5f, 0x5c, 0x20, 0x7c, 0x2e, 0x20,
                        0x20, 0x20, 0x20, 0x20,
27                  0x20, 0x2f, 0x20, 0x2f, 0x20, 0x20, 0x2f, 0x20,
                        0x2f, 0x5f, 0x5f, 0x5f,
28                  0x20, 0x20, 0x5f, 0x5f, 0x5f, 0x20, 0x20, 0x5f,
                        0x5f, 0x5f, 0x5f, 0x20,
29                  0x7c, 0x20, 0x7c, 0x5f, 0x29, 0x20, 0x7c, 0x20,
                        0x28, 0x5f, 0x5f, 0x5f,
30                  0x20, 0x7c, 0x20, 0x7c, 0x20, 0x20, 0x7c, 0x20,
                        0x7c, 0x0a, 0x2e, 0x3c,
31                  0x20, 0x5c, 0x20, 0x2f, 0x20, 0x20, 0x3c, 0x20,
                        0x20, 0x20, 0x20, 0x20,
32                  0x2f, 0x5c, 0x20, 0x20, 0x20, 0x20, 0x3e, 0x20,
                        0x28, 0x20, 0x23, 0x29,
33                  0x20, 0x7c, 0x23, 0x29, 0x20, 0x20, 0x20, 0x20,
                        0x2f, 0x20, 0x2f, 0x20,
34                  0x20, 0x2f, 0x20, 0x2f, 0x20, 0x5f, 0x5f, 0x20,
                        0x5c, 0x2f, 0x20, 0x5f,
35                  0x20, 0x5c, 0x2f, 0x20, 0x5f, 0x5f, 0x20, 0x5c,
                        0x7c, 0x20, 0x20, 0x5f,
36                  0x20, 0x3c, 0x20, 0x5c, 0x5f, 0x5f, 0x5f, 0x20,
                        0x5c, 0x7c, 0x20, 0x7c,
37                  0x20, 0x20, 0x7c, 0x20, 0x7c, 0x0a, 0x20, 0x20,
                        0x7c, 0x20, 0x7c, 0x20,
```

Contenu masqué

```
38                  0x20, 0x20, 0x20, 0x3c, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x2f,
39                  0x5c, 0x20, 0x20, 0x20, 0x2d, 0x2e, 0x20, 0x20,
                        0x20, 0x5f, 0x5f, 0x5c,
40                  0x20, 0x20, 0x20, 0x2f, 0x20, 0x2f, 0x5f, 0x5f,
                        0x2f, 0x20, 0x2f, 0x20,
41                  0x2f, 0x5f, 0x2f, 0x20, 0x2f, 0x20, 0x20, 0x5f,
                        0x5f, 0x2f, 0x20, 0x2f,
42                  0x20, 0x2f, 0x20, 0x2f, 0x7c, 0x20, 0x7c, 0x5f,
                        0x29, 0x20, 0x7c, 0x5f,
43                  0x5f, 0x5f, 0x5f, 0x29, 0x20, 0x7c, 0x20, 0x7c,
                        0x5f, 0x5f, 0x7c, 0x20,
44                  0x7c, 0x0a, 0x20, 0x20, 0x20, 0x5c, 0x20, 0x20,
                        0x20, 0x3c, 0x20, 0x20,
45                  0x3c, 0x20, 0x20, 0x20, 0x56, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x3e,
46                  0x20, 0x29, 0x2e, 0x2f, 0x5f, 0x2e, 0x5f, 0x28,
                        0x5c, 0x20, 0x20, 0x5c,
47                  0x5f, 0x5f, 0x5f, 0x5f, 0x5f, 0x2f, 0x20, 0x2e,
                        0x5f, 0x5f, 0x5f, 0x2f,
48                  0x5c, 0x5f, 0x5f, 0x5f, 0x2f, 0x5f, 0x2f, 0x20,
                        0x2f, 0x5f, 0x2f, 0x20,
49                  0x7c, 0x5f, 0x5f, 0x5f, 0x5f, 0x2f, 0x7c, 0x5f,
                        0x5f, 0x5f, 0x5f, 0x5f,
50                  0x2f, 0x7c, 0x5f, 0x5f, 0x5f, 0x5f, 0x5f, 0x2f,
                        0x20, 0x0a, 0x20, 0x20,
51                  0x2e, 0x29, 0x2f, 0x5c, 0x20, 0x20, 0x20, 0x3c,
                        0x20, 0x20, 0x3c, 0x20,
52                  0x20, 0x2e, 0x2d, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x2f, 0x20, 0x20, 0x5c,
53                  0x5f, 0x27, 0x5f, 0x29, 0x20, 0x29, 0x2d, 0x2e,
                        0x2e, 0x20, 0x20, 0x20,
54                  0x2f, 0x5f, 0x2f, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
55                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
56                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
57                  0x20, 0x20, 0x20, 0x0a, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x5c, 0x20,
58                  0x20, 0x3c, 0x20, 0x20, 0x20, 0x2e, 0x2f, 0x20,
                        0x20, 0x2f, 0x20, 0x20,
59                  0x3e, 0x20, 0x3e, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x2f, 0x2e,
60                  0x5f, 0x2e, 0x2f, 0x0a, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x2f, 0x5c,
61                  0x20, 0x20, 0x20, 0x3c, 0x20, 0x20, 0x27, 0x2d,
                        0x27, 0x20, 0x3e, 0x20,
62                  0x20, 0x20, 0x20, 0x3e, 0x20, 0x20, 0x20, 0x20,
                        0x2f, 0x20, 0x20, 0x20,
```

Contenu masqué

```
63                  0x0a, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x27, 0x2d, 0x2e,
64                  0x5f, 0x20, 0x3c, 0x20, 0x76, 0x20, 0x20, 0x20,
                        0x20, 0x3e, 0x20, 0x20,
65                  0x20, 0x5f, 0x2e, 0x2d, 0x27, 0x0a, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
66                  0x20, 0x20, 0x20, 0x20, 0x2f, 0x20, 0x27, 0x2d,
                        0x2e, 0x5f, 0x5f, 0x5f,
67                  0x5f, 0x5f, 0x5f, 0x2e, 0x2d, 0x27, 0x20, 0x5c,
                        0x0a, 0x20, 0x20, 0x20,
68                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
                        0x20, 0x20, 0x20, 0x20,
69                  0x20, 0x20, 0x5c, 0x2f, 0x0a
70          };
71
72          printf("sizeof openbsd_art = %zu\n", sizeof openbsd_art);
73          return 0;
74  }
```

Retourner au texte.

Liste des abréviations

ABI Application Binary Interface. 58

API Application Programming Interface. 46

ISO International Organization for Standardization. 2