Le Langage C

- Le C en 20 heures
- Le C en 20 heures en pdf FR

Nouveautés du langage C dans sa prochaine version C23

```
*URL:
https://linuxfr.org/news/nouveautes-du-langage-c-dans-sa-prochaine-version-c
23
*Title:   Nouveautés du langage C dans sa prochaine version C23
*Authors: pulkomandy
         * bayo, vmagnin, Yves Bourguignon, Ysabeau, Lawless, xdelatour,
Julien Jorge, nico4nicolas, Cm, Christophe G., Gil Cot et alkino
*Date:    2022-08-07T23:22:02+02:00
*License: CC By-SA
*Tags:    langage_c
*Score:   43
```

Le C est un langage de programmation développé depuis 1972 par Kenneth Thompson, Brian Kernighan et Dennis Ritchie. Il est, au départ, étroitement lié au développement du système UNIX, mais il a par la suite trouvé de nombreuses autres utilisations.

Il a influencé le développement de plusieurs autres langages dont C++, Objective-C, Java, D et C#.

La version C23, qui sera vraisemblablement finalisée en 2023, apporte son lot de nouveautés.

Après un bref historique de la normalisation du langage, cet article parcourt les principaux changements présents dans cette nouvelle version.

C23 - cppreference.com C23 is shaping up: New keywords and fixed assert to land in updated standard C2x - wikipedia C23 is Finished: Here is What is on the Menu - The Pasture C23 implications for C libraries Le "brouillon" de la spécification C23 du 3 septembre 2022

Normalisation

La première version stable du langage est celle publiée en 1978 dans le livre *The C Programming Language*. Les premiers compilateurs ont été implémentés en suivant ce livre, cependant il ne s'agit pas d'une spécification du langage et le comportement des compilateurs n'était pas toujours identique sur des cas inhabituels.

C'est pourquoi un groupe de travail de l'ANSI s'est chargé, à partir de 1983, de rédiger une

spécification plus formelle, qui clarifie toutes les zones d'ombre laissées par le livre de 1978. Cela a pris beaucoup de temps et cette spécification ne sera publiée qu'en 1989. Il faut noter que dès cette version, le C s'inspire du C++ qui est encore un tout jeune langage, et intègre par exemple la notion de prototype de fonction qui n'était pas présente dans les versions précédentes.

La version normalisée du C passe ensuite dans les mains de l'ISO, qui republiera d'abord la norme ANSI, puis deux volumes de « corrections techniques » en 1994 et 1996 pour corriger certaines erreurs mineures dans la spécification et ajouter quelques petits changements.

Mais la première grosse mise à jour du C normalisé arrive en 1999. On trouve en C99 les tableaux de taille variable, les pointeurs restreints, les nombres complexes, les littéraux composés, les déclarations mélangées avec les instructions, les fonctions `inline`, le support avancé des nombres flottants, et une nouvelle syntaxe de commentaire inspirée de C++.

La version suivante, encore plus de dix ans plus tard, est C11, qui ajoute les threads, les expressions à type générique, et une meilleure prise en charge de l'Unicode. Elle rend également optionnelles certaines fonctionnalités qui étaient obligatoires en C++. Par la suite elle est mise à jour par la version C17, qui ne comporte que des clarifications et corrections sans apporter de grandes nouveautés.

Comme toutes les normes publiées par l'ISO, la version finalisée des documents n'est accessible que moyennant paiement. Cependant, le comité de normalisation met à disposition des versions « brouillon » proches de la version finale, qui permettent de se faire une idée très précise du contenu de la norme.

On peut également consulter les « notes », toutes sortes de documents liés au processus de normalisation. Certaines de ces notes contiennent des propositions de changement de la norme, cependant, elles sont soumises à un vote et peuvent être rejetées. Une fois qu'un document est rejeté, il est mal vu de le proposer à nouveau sans d'importantes modifications. Aussi, le travail pour proposer des modifications dans la norme se compose au moins autant de politique et de négociations que de travail pour rédiger la proposition technique.

La proposition de changements est ouverte à tout le monde selon des règles bien documentées. Les propositions sont ensuite revues lors des réunions du comité de normalisation qui choisit, ou pas, de les intégrer dans la norme. Cependant, il est recommandé de prendre contact avec les membres du comité pour arriver à rédiger une proposition qui aura une chance de passer cet examen.

En plus des activités d'intégration de nouvelles fonctionnalités, le comité doit aussi régler des problèmes administratifs et techniques. Par exemple, la spécification du C était écrite à l'aide de troff et avait des problèmes avec les outils modernes pour manipuler ce langage. Il y a donc eu une migration vers LaTeX. De même, en 2020 le groupe de travail a dû se réorganiser pour faire des meetings virtuels, ce qui n'était pas le cas auparavant.

Les trucs supprimés

Définition de fonctions de style « K&R »

N2432

Cette façon de définir des fonctions date d'avant la première version normalisée du C (C89). Elle est

obsolète depuis longtemps, à tel point qu'elle était déjà déclarée obsolète dans la version C89. Elle ressemble à ça :

```
int add(a, b)
    int a;
    int b;
{
    return a + b;
}
```

Cette forme est maintenant interdite et il faudra utiliser la « nouvelle » syntaxe qui indique les types des paramètres dans les parenthèses :

```
int add(int a, int b)
{
    return a + b;
}
```

Déclaration de fonctions sans spécifier les paramètres

N2841

Le code suivant compilait sans problème dans les versions précédentes du C, mais en C23 ce ne sera plus le cas :

```
extern int foo();

void bar(void) {
    int a = 0, b = 1, c = 2;
    foo(a, b, c);
}
```

Le comportement de la déclaration de foo ci-dessus sera équivalent à :

```
extern int foo(void);
```

C'est le même comportement qu'en C++. La déclaration de fonctions sans spécifier les paramètres était déconseillée depuis la version C89, il était donc temps de la retirer du langage.

Fonctions sans arguments fixes

N2975

On peut se demander pourquoi la déclaration sans paramètres n'a pas été supprimée plus tôt. La raison est qu'il existe un cas ou cette syntaxe était utile. En effet, c'était la seule façon de déclarer une fonction pouvant accepter un nombre quelconque de paramètres, y compris aucun.

Pour arriver au même résultat, une autre fonctionnalité du langage a été revue, il s'agit des fonctions variadiques.

Habituellement une fonction variadique s'implémente de cette façon :

```
int printf(int format, ...)
{
    va_list ap;
    va_start(ap, format); // on déclare le dernier argument "connu"

    if (format == INTEGER) {
        int valeur = va_arg(ap, int);
    }

    if (format == DOUBLE) {
        double valeur = va_arg(ap, double);
    }

    va_end(ap);
}
```

On remarque que la macro `va_start` prend en paramètre le nom du dernier paramètre connu. Ce qui empêche d'écrire une fonction de ce type :

```
int foo(...)
{
}
```

La macro `va_start` est modifiée en C23 pour ignorer ce deuxième argument.

On pourra donc désormais écrire :

```
int printf(int format, ...)
{
    va_list ap;
    va_start(ap);

    if (format == INTEGER) {
        int valeur = va_arg(ap, int);
    }

    if (format == DOUBLE) {
        double valeur = va_arg(ap, double);
    }

    va_end(ap);
}
```

Le code existant continuera toutefois de compiler sans erreur, la macro pouvant toujours recevoir un deuxième argument dont elle ne fera rien. Et on pourra utiliser cela pour déclarer des fonctions avec n'importe quel nombre d'arguments, en remplacement de l'ancienne notation.

Il s'agit en quelque sorte d'un retour en arrière, puisque la fonction `va_start()` fonctionnait de cette

façon avant sa normalisation dans `stdarg.h` (on la trouvait alors dans `vararg.h` qui est toujours disponible dans GCC par exemple). Le deuxième argument avait été ajouté pour satisfaire aux besoins de certains compilateurs et exposait un détail d'implémentation et d'ABI qui n'a finalement rien à faire dans la spécification du langage.

Suppression des trigraphes ??!

Le langage C utilise peu de caractères spéciaux afin de pouvoir fonctionner avec n'importe quel code de caractères sur 7 bits. Le jeu de caractères de base est ISO 646 qui ne comporte que 82 caractères invariants (plus 16 caractères de contrôle non imprimables, et 12 caractères variables qui sont remplacés dans la variante de cet encodage adaptée à chaque langue).

Ces 82 caractères sont insuffisants, aussi le C permet de substituer des séquences de trois caractères aux caractères non disponibles (le remplacement est fait par le préprocesseur avant tout autre traitement).

Huit caractères supplémentaires hors ISO 646 sont ainsi pris en charge.

```
// Du code sans trigraphes
#include <stdio.h>

int main(void)
{
   int tab[3];
   return 0;
}
```

```
// Le même avec des trigraphes
??=include <stdio.h>

int main(void)
??<
   int tab??(3??);
   return 0;
??>
```

Cependant, même sur les machines utilisant un encodage ISO 646, cette solution n'était pas vraiment populaire et il était en général possible d'utiliser d'autres caractères de substitution, le code ressemblant par exemple à ceci :

```
// Le même en utilisant l'encodage ISO-646-FR
£include <stdio.h>

int main(void)
é
   int tab°3§;
   return 0;
è
```

Aujourd'hui plus aucun système n'est limité à un encodage du texte sur 7 bits. On utilise donc des encodages basés sur ASCII, ou, dans le pire des cas, EBCDIC, dans lesquels les caractères nécessaires

à l'écriture du C sont tous disponibles. Les trigraphes sont donc supprimés en C23 (cela avait déjà été fait pour C++ dans sa version C++17).

Le fait d'avoir étendu le jeu de caractères nécessaires a également été l'occasion d'ajouter quelques autres caractères obligatoires pour pouvoir stocker du code source : `@`, `$`, et `` ` `` (N2701). Ils ne sont pas utilisés directement par le langage, mais peuvent être utiles par exemple dans les commentaires de type Doxygen :

```
/** @brief Une fonction avec un commentaire de type doxygen.
 * @param[in] a: un paramètre
 * @return: une valeur
 */
int uneFonction(int a);
```

Ou encore pour mettre des adresses e-mails dans des chaînes de caractères ou dans des commentaires. Ces 3 caractères étant présents dans ASCII et dans la plupart des variantes de EBCDIC, ils ont pu être ajoutés sans poser de contraintes supplémentaires.

Les entiers signés sont forcément en complément à deux

Dans l'histoire de l'informatique, les nombres signés n'ont pas toujours été représentés de la même façon. Certaines architectures de processeurs utilisaient par exemple un entier non signé combiné avec un bit de signe séparé. Par exemple, si 1 est représenté par `0 0000001`, -1 est représenté par `1 0000001`.

D'autres utilisent la négation bit à bit du nombre à représenter. On parle alors de complément à un. Par exemple, si 1 est représenté par `00000001`, alors -1 sera représenté par `11111110`.

Le langage C ne prenait pas position sur ce sujet et était indépendant du format de représentation utilisé. Ce qui permettait de faire fonctionner facilement un compilateur C sur n'importe quel processeur, mais avec une conséquence importante : les opérations de conversion entre entiers signés et non signés provoquaient de nombreux cas de comportements indéfinis ou, au mieux, définis par l'implémentation.

Plus aucun processeur moderne n'utilise autre chose que le complément à deux. Cette façon de faire semble moins évidente que les deux autres, mais c'est en réalité la plus simple à mettre en place car elle élimine la plupart des cas particuliers. Les mêmes instructions du processeur peuvent être utilisées pour les nombres positifs et négatifs dans presque tous les cas. C'est donc maintenant le seul format autorisé par le langage C, ce qui a permis de grandement simplifier la spécification du langage et en particulier le comportement des conversions entre types signés et non signés.

Changements sur la gestion des caractères Unicode

Les littéraux préfixés par u ou U sont forcément de l'UTF-16 et de l'UTF-32 (N2728).

Dans les versions précédentes de C, le support de l'Unicode avait d'abord été ajouté avec le type `wchar_t` et les littéraux utilisant le préfixe L. Cependant, aucun encodage spécifique n'était imposé, et donc `wchar_t` pouvait être soit un type sur 16 bits, soit sur 32 bits, et pas forcément en Unicode.

Ensuite, les types `char16_t` et `char32_t` ont été ajoutés, avec les préfixes correspondants u et U. Mais il n'était toujours pas obligatoire d'utiliser de l'Unicode. En C23, c'est désormais le cas, et tout

autre encodage est interdit.

```
int main(void)
{
    wchar_t* exemple1 = L"Une chaîne dans un encodage non spécifié";
    char16_t* exemple2 = u"Une chaîne encodée en UTF-16";
    char32_t* exemple3 = U"Une chaîne encodée en UTF-32";
}
```

De plus, il est interdit de mélanger les différents préfixes :

```
int main(void)
{
    // Code qui n'est plus valide en C23 :
    char32_t* exemple3 = U"Et si on mélangeait" u" les encodages?";
}
```

Changement du comportement de realloc

Le comportement de `realloc` lorsqu'on donne 0 comme deuxième paramètre devient indéfini. Auparavant il était laissé au choix de l'implémentation et pouvait par exemple, faire l'équivalent d'un `free`, ou ne rien faire et conserver le pointeur original. Et la fonction peut aussi, peut-être, mettre à jour `errno`.

La rédaction de la spécification avait déjà été modifiée en C17 suite à des différences constatées entre les différentes implémentations dans différents systèmes lors de la mise à jour de la spécification POSIX par l'Austin Group (sans que la spécification puisse dire qui avait raison). Le problème a été remonté via le numéro 400.

Cependant, la correction qui a été faite et intégrée dans C17 ne résolvait pas complètement le problème, ce qui a été souligné via la demande de clarification N2428.

Il y a deux cas à tester : si le pointeur passé en paramètre est valide, et s'il ne l'est pas. Et il y a trois choses à vérifier en sortie : le pointeur retourné par `realloc`, le fait que la mémoire pointée par le pointeur passé en paramètre a été libérée, et une éventuelle mise à jour de `errno`. On trouvait dans différentes implémentations de C99 un peu tous les comportements possibles dans ces cas.

Finalement, en C23 le comportement de realloc avec une taille de 0 devient indéfini (N2464). Il faudra utiliser `free()` si on veut libérer de la mémoire, et rien si on ne veut rien faire. Comme ça il n'y a plus d'ambiguïtés dans la spécification du langage. De son côté, POSIX avait déjà adopté une définition plus claire du comportement, sur laquelle on pourra compter pour les systèmes compatibles POSIX mais pas forcément ailleurs. Le comportement pour les systèmes POSIX est que l'ancien pointeur n'est pas libéré, et que realloc retourne un pointeur valide vers une zone de 0 octet (qui peut être redimensionnée via de futurs appels à realloc).

```
void* ptr = realloc(NULL, 0); // Utilisation valide de realloc avec une
taille de 0, pour créer un pointeur vers une zone vide
fd = open("data.bin", "r");

while(1) {
    // Récupération d'une taille mémoire à utiliser
```

```
    uint32_t size = 0;
    int ret = read(fd, &size, sizeof(size));
    if (ret == 0) {
        // fin du fichier
        break;
    }

    // Allocation de la mémoire pour lire le bloc (fonctionne même si la
taille du bloc est de 0)
    void* ptr2 = realloc(ptr, size);
    if (ptr2 == NULL) {
        // problème d'allocation, on quitte la boucle pour libérer les
ressources
        break;
    } else {
        // réallocation réussie, on peut oublier l'ancien pointeur et
utiliser le nouveau
        ptr = ptr2;
    }

    // Lecture du bloc
    read(fd, ptr, size);
}

// realloc(ptr, 0); // Utilisation obsolète de realloc pour libérer la
mémoire, qui ne fonctionne plus.
free(ptr); // La bonne façon de faire

close(fd);
return -1;
```

stdbool.h

En C99, une première tentative d'ajout de `bool`, `true` et `false` dans le langage avait été faite, pour assurer la compatibilité avec C++ qui en avait fait un type et deux mots clés du langage.

Cependant, de nombreux programmes C avaient leur propre définition du type bool (par exemple en utilisant un `typedef`) et des valeurs true et false (par un `#define` ou une énumération). En faire des mots clés réservés pour le langage C aurait cassé ces programmes. La solution mise en place a donc été un peu plus compliquée :

- Ajout du type *Bool dans le langage C (ce nom commençant par un* suivi d'une lettre majuscule, par convention, les noms de ce type sont réservés pour l'usage interne du compilateur et donc le code C existant suivant la norme à la lettre ne devrait pas y voir de problème)
- Ajout de l'en-tête stdbool.h qui peut être inclus pour pouvoir utiliser `bool`, `true` et `false` (sous forme de `#define`)

Cette solution est finalement peu pratique, et ne résout pas vraiment le problème de compatibilité avec le C++.

Finalement, en C23, `true` et `false` sont des *constantes prédéfinies*, un concept ajouté au langage C

pour l'occasion (mais utilisé ensuite aussi pour `nullptr`) grâce à la note N2935.

Tout le fichier stdbool.h est donc maintenant obsolète.

Retrait de ,

Le type `bool`, les macros `alignof` et `alignas`, et le qualificateur `thread_local` reçoivent un traitement similaire. Ils ont tous été introduits avec un mot clé commençant par un _ et une majuscule, et divers fichiers d'en-tête ajoutant des macros pour le nom usuel :

```
# define alignas _Alignas
# define alignof _Alignof
# define bool _Bool
# define static_assert _Static_assert
# define thread_local _Thread_local
```

Avec N2934, ces définitions sont désormais disponibles sans avoir besoin d'inclure aucun fichier d'en-tête.

Cela signifie également qu'il n'y a plus besoin de `__alignof_is_defined`, `__alignas_is_defined`, on pourra tester plus directement la présence de ces macros :

```
#ifndef alignof
#error alignof n'est pas défini
#endif
```

Les premières versions de cette note tentaient de transformer ces noms en véritables mots clés réservés du langage, mais cela causait encore trop de problèmes de compatibilité avec le code existant. Peut-être dans une prochaine version du C après quelques dizaines d'années supplémentaires ?

Les fonctionnalités qui deviennent obsolètes

Obsolètes (ou en jargon informatique *deprecated*) signifie que ces fonctionnalités ne sont pas encore supprimées du langage. Mais ce sera probablement fait dans une prochaine version de la norme.

Les macros pour détecter le format des nombres flottants

Le langage C, essayant toujours de pouvoir s'implémenter facilement sur n'importe quelle architecture matérielle et logicielle, n'impose pas de format d'encodage pour les nombres flottants (les types `float` et `double` et depuis C99, leurs équivalents pour les nombres complexes).

Cependant, si une implémentation du langage C utilise le format le plus répandu, elle peut l'indiquer via deux macros. Ainsi le code compilé sur cette implémentation pourra détecter que ce format est utilisé et l'employer pour réaliser certaines optimisations.

Le nom de ces macros était `__STDC_IEC_559__` et `__STDC_IEC_559_COMPLEX__`. Cependant, l'IEC a changé sa numérotation en 1997 pour avoir des numéros compatibles avec ceux de l'ISO. La norme IEC 559 est donc devenue la norme ISO/IEC 60559.

Le C23 introduit donc de nouvelles macros avec le nouveau nom, et en profite pour distinguer le support des nombres à virgule flottante binaire (l'exposant est une puissance de 2) et des nombres à virgule flottante décimaux (l'exposant est une puissance de 10, ce qui évite des erreurs d'arrondi surprenantes pour les humains habitués à réfléchir en base 10) :

- `__STDC_IEC_60559_BFP__` remplace `__STDC_IEC_559__` et indique que les nombres flottants binaires utilisent le format spécifié dans la norme ISO/IEC 60559, et que les fonctions mathématiques de la bibliothèque standard sont implémentées.
- `__STDC_IEC_60559_DFP__` fait de même pour les flottants décimaux,
- `__STDC_IEC_60559_COMPLEX__` remplace `__STDC_IEC_559_COMPLEX__` et fait de même pour les nombres complexes.

Détail amusant, bien que les macros utilisent le nom IEC 60559, la documentation parle de la norme IEEE 754, qui a le même contenu mais est normalisée par l'ANSI et l'IEEE aux États-Unis.

```
#include <stdio.h>
#include <complex.h>
#include <tgmath.h>

#if __STDC_IEC_60559_COMPLEX__

int main(void)
{
    double complex z1 = I * I;     // i^2
    printf("I * I = %.1f%+.1fi\n", creal(z1), cimag(z1));

    double complex z2 = pow(I, 2); // i^2 aussi
    printf("pow(I, 2) = %.1f%+.1fi\n", creal(z2), cimag(z2));

    double PI = acos(-1);
    double complex z3 = exp(I * PI); // La formule d'Euler: e^i*pi=-1
    printf("exp(I*PI) = %.1f%+.1fi\n", creal(z3), cimag(z3));
}

#else

#error Les nombres complexes ne sont pas représentés dans le format spécifié
par ISO/IEC 60599

#endif
```

DECIMAL_DIG

N2108

La constante `DECIMAL_DIG` a été ajoutée en C99 et donne le nombre de décimales nécessaire pour représenter un nombre de type `long double` sans perdre de précision. Cependant, il existe déjà une constante `LDBL_DECIMAL_DIG` avec la même valeur (ainsi que des constantes équivalentes pour les autres types de nombres à virgule flottante).

DECIMAL_DIG est donc inutile, et devient obsolète en C23.

```
long double unNombre;
printf("%.*Lf", LDBL_DECIMAL_DIG, unNombre);
```

Définitions de macros redondantes dans math.h

Les définitions de `INFINITY`, `DEC_INFINITY`, `NAN` et `DEC_NAN` étaient disponibles à la fois dans `<math.h>` et `<limits.h>`. Désormais, seul ce dernier pourra être utilisé.

Les nouvelles fonctionnalités

Le retour des ptrdiff_t sur 16 bits

N2808 est un retour en arrière sur un changement intervenu dans C99.

Le type `ptrdiff_t` permet de calculer la différence entre deux pointeurs. Pour pouvoir traiter tous les cas, ce type doit forcément être signé (la différence entre deux pointeurs peut être négative). Cependant, en C99, le type `ptrdiff_t` devait pouvoir représenter des valeurs allant jusqu'à 65535, et il nécessitait donc au moins 17 bits.

Cela n'est pas un problème sur les systèmes utilisant des pointeurs plus gros, par exemple sur 32 bits. Mais c'est plus gênant pour les microcontrôleurs (AVR, STM8…) sur lesquels il est souhaitable d'économiser la mémoire autant que possible. Il est donc dommage de forcer ces architectures à avoir un type `ptrdiff_t` plus large que le type des pointeurs.

Ces implémentations sont donc autorisées à utiliser un type 16 bits pour `ptrdiff_t`, et en conséquence, une allocation mémoire contiguë ne devrait jamais dépasser 32767 octets.

```
char data[40000];
ptrdiff_t size = &data[39999] - &data[0]; // peut-être interdit si vous êtes
sur un système 16 bits
```

Distinction entre les tableaux à taille variable et les types modifiés par des variables

N2778

Le C99 a ajouté la possibilité de déclarer des tableaux dont la taille n'est pas connue à la compilation :

```
void fonction(int n)
{
    int tableau[n]; // alloué sur la pile de façon similaire à la fonction
non-standard alloca()
}
```

Ce type d'allocation n'est pas possible dans tous les cas, aussi, une implémentation du C peut choisir de ne pas autoriser ce code. Dans ce cas la macro `__STDC_NO_VLA__` doit être définie.

Une idée proche de celle des tableaux à taille variable est les types modifiés par une variable, par

exemple dans ce cas:

```
void foo(int n, double (*x)[n])
{
    (*x)[n] = 1;
}
```

Il n'y a pas d'allocation dynamique dans ce cas (le tableau est juste passé en paramètre), mais on indique au compilateur (via le [n] dans le paramètre de la fonction) quelle est la taille du tableau. Cela permet de détecter les tentatives d'accès au tableau à des index hors de ses limites, à la compilation ou à l'exécution du code.

Cette syntaxe était auparavant interdite si la macro __STDC_NO_VLA__ était définie. Désormais les deux concepts sont séparés et les types modifiés par des variables sont autorisés, même si les tableaux à taille variable ne le sont pas.

En conséquence, les types modifiés par une variable sont maintenant une fonctionnalité obligatoire dans les implémentations standards du C.

nullptr (comme en C++)

Un défaut historique du C est que la constante NULL peut être définie soit comme un entier, soit comme un `void*`. De plus, c'est seulement une constante définie par le préprocesseur (via un `#define`), ce qui empêche de se reposer dessus dans les étapes suivantes de la compilation.

Ces problèmes sont bien connus et ont déjà été résolus dans C++11 par l'ajout de `nullptr` et du type `nullptr_t`. Ces changements ont été reportés dans la spécification du C par la note N3042.

Amélioration des énumérations

En C, les énumérations sont implémentées avec un type sous-jacent. Par exemple on peut écrire ceci:

```
enum {
   uneValeur = 4;
};

int unEntier = uneValeur;
```

La norme ne spécifie pas un type fixe, et le compilateur peut choisir comme il veut. Par exemple, toutes les énumérations peuvent être des entiers, ou bien un type est choisi en fonction des valeurs utilisées dans l'énumération et du nombre de bits nécessaires pour les représenter.

Le problème est que les constantes qui peuvent être indiquées dans une enum sont, d'après la norme, forcément de type `int`. Ceci est donc interdit:

```
enum a {
    a0 = 0xFFFFFFFFFFFFFFFFULL // il ne s'agit pas d'un int, mais d'un
unsigned long long
};
```

En pratique, la plupart des compilateurs autorisent cette notation et il n'y a pas de problème. D'ailleurs, c'est autorisé dans les normes C++.

La note N3029 propose donc d'autoriser cette notation pour définir une énumération avec des valeurs en dehors de l'intervalle INT_MIN - INT_MAX.

On peut penser que cet intervalle est largement suffisant, mais il ne faut pas oublier les implémentations du C où le type `int` est un type sur seulement 16 bits, une limite qui peut être rapidement atteinte. Maintenant, ces implémentations du C pourront avoir des `int` sur 16 bits, mais des valeurs d'énumérations allant au-delà si nécessaire.

La note N3029 est assez longue, car elle prend beaucoup de précautions pour éviter des problèmes de compatibilité, y compris avec l'implémentation déjà en place dans différents compilateurs C pour ce type de code.

Ceci laisse quand même un problème assez gênant : c'est toujours le compilateur qui choisit le type d'une énumération. Cela rend difficile l'écriture de code portable sur de nombreux compilateurs et architectures.

La note N3030 apporte donc une nouvelle syntaxe pour pouvoir forcer un type spécifique :

```
// Une énumération explicitant le type sous-jacent
enum a : unsigned long long {
    a0 = 0xFFFFFFFFFFFFFFFFULL
};
```

Il s'agit, ici encore, d'une amélioration déjà existante en C++ qui a été récupérée dans la norme C.

On peut également obtenir une erreur à la compilation si l'une des valeurs énumérées ne peut pas être stockée dans le type choisi:

```
enum a : uint16_t {
    a0 = 0x10000 // erreur: il faudrait 17 bits pour stocker cette valeur et
le type uint16_t n'en a que 16
};
```

Améliorations des macros variadiques

En C (normalisé dans C99 mais cela existait déjà dans la plupart des compilateurs), on peut déclarer des macros avec un nombre d'arguments variable :

```
// La macro LOG appelle fprintf avec stderr comme premier argument, suivi de
tous les autres arguments sans modification
#define LOG(X, ...) fprintf(stderr, X, __VA_ARGS__)
```

Cependant, ce système de macros est contre-intuitif, surtout lorsqu'on ne veut pas utiliser les arguments optionnels:

```
LOG("Valeur de x: %d\n", x);
    // remplacé par fprintf(stderr, "Valeur de x: %d\n", x);
    // aucun problème
```

```
LOG("Bonjour\n");
    // remplacé par fprintf(stderr, "Bonjour\n",);
    // remarquez la virgule supplémentaire, ce code ne compile pas!
```

La plupart des compilateurs C proposent une façon de contourner ce problème. Les compilateurs Microsoft Visual C++ et Borland/Embarcadero C++ suppriment la virgule indésirable, ignorant ce qui est écrit dans la norme.

GCC implémente __VA_ARGS__ conformément à la norme mais propose pas moins de trois autres options pour obtenir la suppression de la virgule :

```
// Utilisation de ## pour faire « disparaître » la virgule lorsque
__VA_ARGS__ est vide
#define LOG(X, ...) fprintf(stderr, X, ## __VA_ARGS__)

// Une syntaxe différente qui permet de
#define LOG(args...) fprintf (stderr, args)

// __VA_OPT__ (introduit en C++20, mais accepté aussi dans le code C par
gcc)
#define LOG(format, ...) fprintf (stderr, format __VA_OPT__(,) __VA_ARGS__)
```

Finalement, la note N3033 normalise cette dernière option avec __VA_OPT__. Cette note a été intégrée à la fois dans C++20 et C23, afin de conserver une cohérence entre les deux langages et de ne pas se retrouver avec deux façons différentes de faire la même chose.

Spécificateurs de stockage pour les littéraux composites

Le code suivant n'est pas valide (mais il est accepté sans problème par gcc) :

```
int function(void) {
   static struct foo  x =  (struct foo) {1, 'a', 'b'};
}
```

Le « problème » est l'initialisation d'une variable `static` à partir d'une structure qui n'est pas déclarée comme une constante.

On peut essayer de déclarer le littéral composite (la partie à droite du =) comme une constante :

```
static struct foo  x =  (constexpr struct foo) {1, 'a', 'b'};
```

Ce n'était pas autorisé dans les versions précédentes du C. La note N3038 autorise cette notation. Cela est valable pour tous les spécificateurs de stockage: `constexpr`, `const`, `static` et `thread_local`.

Cela permet des simplifications d'écriture pour des choses plus complexes, par exemple, le code suivant :

```
// Déclare un pointeur statique sur une stucture elle-même statique
static struct foo* p = &(static struct foo) {1, 'a', 'b'};
```

remplace, de façon plus compacte, ce qu'il fallait écrire dans les versions précédentes du C :

```
static struct foo  Unique =  (struct foo) {1, 'a', 'b'};
static struct foo* p      =   &Unique;
```

constexpr

Puisqu'on parle de `constexpr`, il s'agit encore une fois d'un mot-clé récupéré du C++ (du C++11 pour être précis).

L'idée est de permettre de déclarer des constantes dans d'autres types que `int`. On pourrait penser que `const` serait suffisant pour déclarer des constantes, mais ce n'est pas exactement le cas. Une variable avec un type `const` ne peut pas être modifiée, mais il s'agit tout de même d'une variable.

En particulier cela signifie que le code suivant est valide :

```
const int a = 47;
const int* b = &a; // a est une variable, elle est stockée quelque part en
mémoire et a donc une adresse.
```

Avec `constexpr`, ajouté par la note N3018 on a véritablement une constante qui n'est pas une variable :

```
constexpr int a = 47;
constexpr int* b = &a; // interdit: une constante n'a pas d'adresse

// de la même façon que c'est interdit d'écrire:
constexpr int* c = &47;
```

Cela a des conséquences par exemple sur l'allocation des tableaux :

```
const int a = 47;
int b[a]; // crée un tableau de taille variable (VLA)
static int c[a]; // interdit : a n'est pas une constante et un tableau
static ne peut pas être de taille variable
```

```
constexpr int a = 47;
int b[a]; // crée un tableau de taille fixe, 47 éléments
static int c[a]; // crée un tableau de taille fixe
```

La note N2713 clarifie cependant que le compilateur peut choisir, dans le cas particulier des tableaux, que le compilateur est autorisé à aller plus loin que la définition stricte d'une expression s'il arrive à calculer une valeur lors de la compilation. Mais on ne peut pas être sûr que tous les compilateurs seront d'accord sur le sujet.

Le nouveau mot clé `constexpr` ouvre également la possibilité de déclarer des constantes qui ne sont pas des types primitifs. Auparavant, la seule façon d'avoir une constante en C était d'utiliser une valeur littérale :

```
1234 // une constante de type int
```

```
1234U // une constante de type unsigned int
1234LU // une constante de type long unsigned int
```

Impossible d'utiliser cette façon de faire pour déclarer une constante de type `uint16_t` ou `size_t` de façon portable, par exemple. Ce code peut donc être compilé sans erreur :

```
const uint16_t entier = 0x10000; // il faudrait 17 bits, mais le compilateur
ne voit pas de problème à faire une conversion implicite.

constexpr uint16_t entier = 0x10000; // ici le compilateur peut générer une
erreur
```

Les possibilités de `constexpr` en C s'appliquent uniquement aux valeurs littérales et à la déclaration de constantes. Cela est beaucoup plus limité que le mot-clé de C++, qui peut quant à lui être utilisé dans des paramètres de fonctions, ou dans des *templates.*

Meilleure gestion des nombres en binaire

En C, on peut utiliser trois bases différentes pour écrire une valeur entière :

```
int decimal = 10; // En décimal
int octal = 010; // En octal, représente la valeur 8 en décimal
int hex = 0x10; // En hexadécimal, représente la valeur 16 en décimal
```

Il manquait le binaire. La note N2549 le rajoute, avec le préfixe `0b` :

```
int binaire = 0b10; // Représente la valeur 2 en décimal
```

La note N2630, quant à elle, ajoute de nouveaux spécificateurs de format pour `printf`, `scanf` et les fonctions associées pour permettre l'affichage et la lecture de nombres en binaire :

```
printf("La valeur %d s'écrit en hexa: %#x et en binaire: %#b\n", valeur,
valeur, valeur);
```

La note N3022 apporte une façon standardisée de connaître le boutisme du système (stockage des nombres en commençant par l'octet de poids fort ou de poids faible), ainsi que des fonctions de manipulation binaire (rotation de bits, comptage du nombre de bits à 1 ou à 0 dans un nombre, etc). Ces fonctions étaient déjà disponibles pour la plupart dans gcc sous un autre nom (par exemple la fonction `__builtin_clz` de gcc est équivalente à `stdc_leading_zeros`).

Cependant, ces opérations ne permettent toujours que de travailler avec des entiers de taille « classique »: 8, 16, 32 ou 64 bits. Que faire si on a besoin d'une autre taille ?

Dans ce cas il faut utiliser les ajouts de la note N2709, qui permet de déclarer des entiers de taille variable:

```
int unEntier = 12; // un entier signé, avec un nombre de bits non normalisé
(rien de nouveau ici)
_BitInt(4) unAutreEntier = 5; // un entier signé sur 4 bits
const unsigned _BitInt(4) encoreUn = 5; // un entier non-signé const sur 4
```

```
bits
_BitInt(4)* unPointeur = &unAutreEntier; // cela fonctionne comme n'importe
quel autre type
```

La note N2775 ajoute les suffixes « wb » et « uwb » pour les littéraux, ce qui permet de déclarer des littéraux de type _BitInt. En particulier c'est utile pour des types avec plus de bits que les types standards, pour lesquels un littéral classique (même avec un suffixe `ull`) ne suffirait pas:

```
unsigned _BitInt(128) unGrandEntier = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFuwb;
```

Chaque implémentation doit définir une valeur pour `BITINT_MAXWIDTH` indiquant quel est le nombre de bits maximal utilisable. Il faudra voir si des entiers de 128 bits (ou plus) sont autorisés, ce qui serait utile par exemple pour certains algorithmes cryptographiques, ou si les compilateurs choisiront de ne pas autoriser plus que 64 bits.

Macros pour les calculs avec vérification de débordement

En C, il n'y a pas de vérification de débordement sur les entiers:

```
uint32_t a = 0xFFFFFFFF;
uint32_t b = 0xFFFFFFFF;
uint32_t c = a + b; // pas d'errreur, ni à la compilation ni à l'exécution
```

La note N2683 ajoute des macros permettant de faire des calculs avec vérification d'erreurs :

```
#include <stdckdint.h>

uint32_t a = 0xFFFFFFFF;
uint32_t b = 0xFFFFFFFF;
uint32_t c;
if (ckd_add(&c, a, b)) {
    // il y a un overflow. Le contenu de c est égal au résultat modulo 32.
}
```

Les deux macros `ckd_sub` (pour les soustractions) et `ckd_mul` (pour les multiplications) fonctionnent de la même façon. Il est possible d'utiliser des types différents pour les 3 paramètres, y compris les `_BitInt` présentés dans le paragraphe précédent.

Cette syntaxe est assez lourde, surtout si on veut réaliser plusieurs opérations (`a = b * c + d` par exemple). La note 2683 propose des améliorations dans des paragraphes *future directions* et des annexes de la norme. On peut donc s'attendre à des évolutions dans les prochaines versions du langage.

Nombres décimaux à virgule flottante

Le langage C ne spécifie pas exactement comment sont traités les nombres à virgule flottante (float, double, et long double). Lorsque les premières versions du C ont été écrites, il n'existait pas de standard pour cela, et les fabricants de processeurs (ou les développeurs de compilateurs, si le processeur n'avait pas de matériel dédié aux calculs flottants) pouvaient développer leur propre format.

Cela a changé depuis, et on peut au moins supposer que les types utilisés sont basés sur une virgule flottante binaire, c'est-à-dire que les nombres sont encodés sous la forme :

(−1)sign x significand x 2^exponent

- 1 bit de signe (positif ou négatif)
- une mantisse comprise entre 0 et 1
- un exposant (positif ou négatif), indiquant par quelle puissance de 2 il faut multiplier le nombre

L'avantage de cette approche est que la multiplication par une puissance de 2 est facile à réaliser (il s'agit d'un simple décalage de bits vers la gauche ou vers la droite). L'inconvénient est que certaines valeurs décimales simples ne peuvent pas être représentées de façon correcte.

Par exemple, le nombre décimal 0.1 serait représenté par 3602879701896397 * 2^-55 = 0.1000000000000000055511151231257827021181583404541015625.

Il y a donc des erreurs d'arrondi là où on ne s'y attend pas en réfléchissant en nombres décimaux.

Une solution est l'utilisation de nombres flottants décimaux, plus faciles à manipuler pour les humains mais moins pour les ordinateurs. C'est ce que propose la note N1724 avec les nouveaux types `_Decimal32`, `_Decimal64` et `_Decimal128` (ce dernier étant optionnel). Avec ces types les nombres sont représentés sous la forme:

(−1)sign x significand x 10^exponent

(−1)sign x significand x 10^exponent

L'ajout de ces types s'accompagne de nombreuses mises à jour dans la bibliothèque standard: les fonctions de formatage (printf, scanf…) doivent pouvoir afficher et décoder ces nombres, les fonctions mathématiques (trigonométrie, logarithmes…) doivent pouvoir les manipuler. Les opérateurs classiques (+, -, =…) doivent fonctionner avec. De nombreuses constantes sont également ajoutées (pour les valeurs minimales et maximales qui peuvent être représentées, par exemple).

On peut également déclarer des variables de ces types:

```
_Decimal32 dixieme = 0.1df;
```

L'apparition de ces types entraîne l'apparition de fonctions pour les manipuler. Par exemple, pour calculer un cosinus dans différents types, on dispose de:

```
float cosf(float v);
double cos(double v);
_Decimal32 cosd32(_Decimal32 v);
_Decimal64 cosd64(_Decimal64 v);
_Decimal128 cosd128(_Decimal128 v);
```

Ce changement s'applique à toutes les fonctions manipulant des nombres à virgule flottante, par exemple les fonctions de manipulation d'exposants : `quantizedN`, `samequantumdN`, `quantumdN`, `llquantexpdN`, ou les fonctions de formatage : `encodedecdN`, `decodedecdN`, `encodebindN`, `decodebindN`, `strfromdN` et de conversion de chaînes en entier: `strtodN`.

Enfin, les fonctions de formatage (de la famille `printf` et `scanf`) peuvent également manipuler ces

nombres, grâce aux spécificateurs "%H", "%D", et "%DD" pour les nouveaux types décimaux flottants _Decimal32, _Decimal64, et _Decimal128 respectivement.

```
_Decimal32 d1;
_Decimal64 d2;
_Decimal128 d3;

printf("Voici des nombres: %H, %D, %DD\n", d1, d2, d3);
```

La note N2341 donne un bon aperçu de toutes les modifications liées aux nombres flottants.

La prise en charge d'Unicode

Le type char8_t pour l'UTF-8

Historiquement en C, les chaînes de caractères sont stockées en utilisant le type `char`. Ce type est prévu pour représenter des caractères en ASCII, et donc sur seulement 7 bits. Par conséquent, certains compilateurs (et certaines architectures de processeur) choisissent que char peut être un type signé ou non signé.

Au départ, l'utilisation d'un type signé permettait d'avoir des fonctions retournant soit un caractère ASCII valide, soit -1 en cas d'erreur. Mais par la suite, cela a posé beaucoup de problèmes pour utiliser d'autres encodages de caractères. Certaines fonctions ont été modifiées pour utiliser un int plutôt qu'un char afin de limiter les problèmes.

De plus, le type char utilise en principe l'encodage « natif » du système, dans le cas d'un système UNIX cela dépend donc de la locale choisie par l'utilisateur (ou par le programme lui-même avec la fonction `setlocale()`). Le comportement d'un programme peut donc varier en fonction de la locale, en plus de l'architecture du processeur et du compilateur. Il devient rapidement très complexe de penser à tous les cas.

Enfin, les fonctions classiques du C pour caractériser un caractère (isalpha, isdigit…) ne peuvent pas fonctionner avec du texte encodé en UTF-8 où un seul caractère peut être encodé sur plusieurs octets.

La note N2653 introduit un nouveau type char8_t qui est dédié au stockage de chaînes en UTF-8. Ce type est forcément non signé.

De plus, les littéraux préfixés par u8 sont maintenant de ce type (c'étaient auparavant des `unsigned char`):

```
char8_t* une_chaine = u8"Caractères accentués encodés en UTF-8";
char* une_autre = "Caractères accentués encodés selon la 'locale' active";
```

Le type char8_t est également disponible en C++, avec quelques différences pour garder une correspondance avec les types char16_t, wchar_t et char32_t qui ont été normalisés de façon un peu différente dans les deux langages.

La note ajoute également les fonctions `mbrtoc8()` et `c8rtomb()` qui s'utilisent comme les fonctions équivalentes pour les autres types « wide char » et permet de convertir les chaînes entre les différentes représentations. Enfin, un type `atomic_char8_t` est également ajouté, si jamais vous

avez besoin d'opérations atomiques (non interruptibles) sur les caractères.

Changements du comportement de \u

La note N2828 modifie la séquence d'échappement \u utilisée pour entrer des caractères unicode dans une chaîne littérale:

```
    const char meow[] = u8"\U49584958";
```

Le code ci-dessus essaie de définir un caractère Unicode de 32 bits. En réalité, la norme ISO 10646 n'autorise pas plus de 21 bits (ce qui permet d'encoder le caractère en UTF-8 sur 4 octets maximum). La note interdit donc officiellement les caractères dépassant cette limite (c'était déjà le cas dans certains compilateurs).

Séparateur de chiffres dans les constantes

La Note N2626 est encore une fonction empruntée au C++ (C++14 pour être précis): les nombres littéraux peuvent être écrits avec un séparateur entre les chiffres:

```
uint32_t a = 0xFFFF'FFFF;
int b = 1'000'000;
uint16_t c = 0b1111'1111'1111'1111;
```

Cela apporte plus de lisibilité, en particulier quand il y a beaucoup de chiffres. Les apostrophes peuvent être placées où on veut, selon ce qu'on veut mettre en valeur.

typeof()

`typeof` permet d'obtenir le type d'une variable. Une utilisation classique pourrait être :

```
// Une macro pour échanger deux valeurs qui ne fonctionne que pour les
entiers :
#define echange_entier(a, b) { int tmp = a; a = b; b = tmp; }

// Une macro adaptée pour fonctionner avec n'importe quel type :
#define echange(a, b) { typeof(a) tmp = a; a = b; b = tmp; }
```

`typeof` est déjà implémenté par la plupart des compilateurs C, mais n'était pas encore normalisé. C'est maintenant le cas avec la note N2927.

Les utilisateurs de C++ ont peut-être remarqué que le mot-clé choisi n'est pas le même qu'en C++, en effet ce dernier utilise `decltype` pour le même usage. La raison de cette divergence est une différence de comportement.

Imaginons ceci dans un fichier .h qui peut être utilisé à la fois par du code C et du code C++ :

```
int a;

typeof(a)   // int en C
decltype(a) // int en C++, jusque-là tout va bien…
```

```
// mais si on ajoute un peu trop de parenthèses:
decltype((a)) // int& en C++ (référence sur un entier)
typeof((a)) // toujours un int en C, bien entendu
```

Si les deux opérations avaient le même nom, on se retrouverait avec des types différents selon que le code est compilé en C ou en C++. Une proposition a été soumise au commité de normalisation du C++ pour ajouter `typeof`, avec une implémentation équivalente à `std::remove_reference_t<decltype(T)>`.

La fonction call_once devient obligatoire

`call_once` a été introduite en C11 dans la partie concernant les threads. L'idée est de s'assurer qu'une fonction est appelée une seule fois, y compris lorsqu'il y a plusieurs threads.

```
void initialise(void)
{
    // Initialisation qui ne sera faite qu'une fois
}

int fonction(void)
{
    static once_flag flag = ONCE_FLAG_INIT;
    call_once(&flag, initialise);
}
```

Cependant, cette construction n'est pas nécessairement limitée aux programmes utilisant plusieurs threads. L'utilisation de la fonction ci-dessus peut très bien se faire plusieurs fois depuis le même thread.

La note N2840 déplace donc `call_once` dans la norme pour la rendre obligatoire y compris pour les implémentations du C ou il n'y a pas de threads. Cela fournit un pendant à `atexit`, qui peut être utilisé pour la dé-initialisation des ressources lors de l'arrêt du programme.

unreachable()

La note N2826 ajoute la fonction `unreachable()` qui permet d'indiquer qu'une branche du code ne peut pas être atteinte.

Cela ressemble un peu à `assert()`, mais cette dernière peut être désactivée avec `#define NDEBUG`, et dans ce cas, il n'y a plus d'indication au compilateur que le code n'est pas supposé être atteignable :

```
int fonction(int param)
{
    assert(param > 0);

    // dans le code écrit ici, le compilateur peut supposer que param est
positif,
    // sauf si le code est compilé en mode NDEBUG. Les versions "debug" et
"release"
    // ont donc peut-être un comportement différent
```

```
}

int fonction2(int param)
{
    if (param <= 0)
        unreachable();

    // dans le code écrit ici, le compilateur peut supposer que param est
positif,
    // dans tous les cas
}
```

__has_include (comme en C++)

Les évolutions de la bibliothèque standard (ou d'autres bibliothèques) rajoutent parfois de nouveaux fichiers .h. Dans certains cas il est souhaitable d'écrire du code pouvant utiliser ces fichiers seulement s'ils sont disponibles.

Historiquement, il fallait pour cela passer par un test avant la compilation (par exemple avec un script « configure », éventuellement généré à l'aide des autotools). Le C++17 a ajouté `__has_include` qui permet de tester la présence d'un fichier directement depuis le préprocesseur. Le C23 intègre cette même fonctionnalité avec N2799

```
#if __has_include(<optional.h>)
 #include <optional.h>
 #define have_optional 1
#elif __has_include(<experimental/optional.h>)
 #include <experimental/optional.h>
 #define have_optional 1
 #define have_experimental_optional 1
#else
 #define have_optional 0
#endif
```

Inférence de type

Normalement, en C, il faut indiquer le type de chaque variable que l'on déclare :

```
double x = 0; // une variable de type double
double y = cos(x); // une autre variable de type double
```

Cela peut rendre le code compliqué par exemple lors de l'écriture de macros ou de code utilisant _Generic :

```
#define swap(a, b) { int tmp = a; a = b; b = tmp; } // macros ne
fonctionnant que pour le type int
```

Ce problème a été réglé en C++11 avec le recyclage du mot clé `auto`:

```
auto y = cos(x); // la fonction cos retourne un double, donc y est de type
double
#define swap(a, b) { auto tmp = a; a = b; b = tmp; } // tmp est du même type
que a, cette macro peut fonctionner avec n'importe quel type
```

Ce mot clé a désormais le même usage en C, suite aux notes N3006 et N3007.

Cela remplace l'utilisation (pré)historique de `auto` pour déclarer des variables locales dans une fonction:

```
void fonction(void)
{
    static int a; // une variable statique (valeur conservée entre les
appels de la fonction, stockée dans un endroit fixe en mémoire)
    auto int b; // une variable automatique, stockée sur la pile
    int c; // une variable automatique aussi, stockée sur la pile
}
```

Plus personne n'indiquait dans son code les variables automatiques explicitement. Le C++ avait donc choisi de recycler ce mot-clé déjà réservé par le langage mais désuet. Le C a fait le même choix pour assurer plus de compatibilité avec le C++.

Initialisation avec

En C, une variable locale déclarée sans initialisation n'est pas initialisée:

```
void fonction(void)
{
    int a; // variable non initialisée
}
```

Il est vivement recommandé de mettre une valeur connue:

```
void fonction(void)
{
    int a = 0;
}
```

Cela se complique un peu pour les structures:

```
typedef struct {
    int premier;
    int deuxieme;
    int troisieme;
} structure;


void fonction(void)
{
    structure a = { 0 }; // initialise tous les champs à 0
```

```
    structure a = { 0, 0, 0 }; // initialise tous les champs à 0
    structure a = { 0, 0 }; // erreur de compilation: le troisième champ
n'est pas initialisé
}
```

Et cela devient encore plus compliqué quand il y a des structures imbriquées. Certains développeurs ont parfois renoncé à comprendre l'utilisation de cette initialisation à 0 (surtout sur les cas complexes mêlant structures, unions et tableaux) et utilisent un memset :

```
    structure a;
    memset(&a, 0, sizeof(a));
```

avec le risque de se tromper dans l'ordre des arguments de la fonction memset, ou d'oublier d'initialiser une variable. Parfois également des développeurs choisissent de désactiver les avertissements du compilateur sur certaines initialisations.

La note N2900 introduit une nouvelle syntaxe:

```
    structure a = { }; // initialise tous les champs à 0
```

C'était déjà autorisé par la plupart des compilateurs, maintenant cela fait partie de la norme. Et cela évite la confusion avec l'initialisation par des valeurs pour chaque champ.

Attributs

Les attributs permettent de manipuler les alertes du compilateur pour certains cas d'usages.

Ils existaient déjà sous diverses formes non standardisées, par exemple `__attribute__(())` pour gcc ou `__declspec()` chez Microsoft. Désormais une nouvelle syntaxe avec deux paires de crochets fait partie du standard : `[[ ... ]]`.

À l'intérieur des crochets, certains attributs sont standardisés, mais chaque implémentation peut ajouter les siens avec un système d'espaces de noms. Par exemple on pourra trouver `[[gnu::always_inline]]` dans du code compilable avec gcc.

La macro `__has_c_attribute( attribute-token )` permet de tester la présence d'un attribut pour savoir si on peut l'utiliser dans le code.

Quelques-uns des attributs standards en C23:

- `[[deprecated]]`, permet de décorer tout un tas d'éléments du code comme dépréciés. Et de lever une alerte s'ils sont utilisés.
- `[[fallthrough]]`, permet d'annoter dans un `switch`, les successions de `case` considérées comme anormales, afin d'éviter les alertes.
- `[[maybe_unused]]`, permet d'annoter une variable qui est certainement utilisée. Par exemple, en fonction de directives de préprocesseur, un paramètre de fonction peut ne plus être utilisé. Mais reste utile dans une autre configuration.
- `[[nodiscard]]`, permet de s'assurer que la valeur retournée par une fonction est bien utilisée. Par exemple pour `malloc()`.
- `[[noreturn]]`, permet d'indiquer qu'une fonction ne se termine pas. C'est le cas par exemple de `abort()`, `exit()` ou `longjmp()`.

Cette syntaxe est la même qui a déjà été standardisée pour C++. Cependant, C11 avait déjà introduit des attributs avec une syntaxe plus simple, ces derniers deviennent obsolètes (N2764).

Il y a quelques subtilités : en C11, le mot-clé ajouté était `_Noreturn` (commençant par un _ pour ne pas empiéter sur des noms de variables, macros, et définis dans le code utilisateur). L'en-tête `stdnoreturn.h` permettait d'utiliser `noreturn` sous forme d'un `#define`.

Dans la nouvelle syntaxe, le problème d'espace de noms ne se pose plus, puisque les attributs sont dans un espace de noms séparé. Cependant, si `stdnoreturn.h` est inclus, le préprocesseur va remplacer `noreturn` par `_Noreturn` et ce y compris dans les nouvelles déclarations. Le C23 accepte donc _Noreturn comme nom d'attribut afin que du code se trouvant dans cette situation continue à compiler.

```
// Fonction noreturn en C11 (notation désormais obsolète)
_Noreturn void f () {
    abort();
}

// Fonction noreturn en C23 (nouvelle notation)
[[noreturn]] void f () {
    abort();
}
```

Fonction avec paramètre anonyme

La définition suivante est désormais valide en C23 :

```
int f(int, int)
{
    return 7;
}
```

Cela permet de définir une fonction qui n'utilise pas certains de ses paramètres. C'est utile si la fonction doit avoir un nombre spécifique de paramètres, par exemple si elle est utilisée via un pointeur de fonction.

Auparavant cette notation était interdite par le standard, il fallait forcément nommer les paramètres et utiliser un attribut (`maybe_unused`) ou d'autres astuces pour éviter d'avoir un avertissement pour un paramètre non utilisé.

Tableaux et éléments similairement qualifiés

En C, les types peuvent être *qualifiés*, l'exemple le plus connu est l'utilisation de la qualification `const` pour les constantes :

```
int a; // un entier variable
const int b; // un entier constant
```

Des complications apparaissent avec les pointeurs: le pointeur lui-même peut être constant (ou pas), et les éléments pointés également:

```
const char* c = "Bonjour"; // un pointeur sur une chaîne constante
c[1] = 'a'; // interdit, le type pointé est constant
c = "Au revoir"; // autorisé, le pointeur n'est pas constant donc il peut
pointer ailleurs

char* const d = strdup(c); // un pointeur constant vers une chaîne variable
d[1] = 'a'; // autorisé, le type pointé n'est pas constant
d++; // interdit, le pointeur est constant et ne peut pas être modifié

const char* const e; // un pointeur constant sur un une chaîne constante
const char* const * const f; // un pointeur constant sur un pointeur
constant sur... et ainsi de suite
```

Cependant, cela ne fonctionne pas pour les tableaux.

```
const int a[] = {1, 2, 3}; // un tableau contenant des entiers constants
```

Il n'y a pas d'endroit où mettre un `const` pour qualifier le tableau lui-même. Cela peut se comprendre, car de toutes façons on ne peut pas affecter ou incrémenter un tableau:

```
a = { 5 }; // interdit
a++; // interdit aussi
```

Dans des cas un peu plus complexes, on se retrouve avec des choses qui ne fonctionnent pas de façon inattendue :

```
// Une fonction de transposition de matrice.
// Le paramètre 'in' n'est pas modifié, il est donc déclaré 'const'
void transpose(int N, int M, double out[M][N], const double in[N][M])
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            out[j][i] = in[i][j];
}

const double a[2][2] = { ... };
double o[2][2];
transpose(2, 2, o, a); // le tableau a est constant et peut être passé en
paramètre

double b[2][2];
transpose(2, 2, o, b); // type incompatible
```

Ou encore, lors de la conversion d'un tableau en pointeur le const n'est pas conservé comme il faudrait :

```
const int foo[5];
memset(&foo, 0, sizeof foo); // écrit dans un tableau qui devrait être
constant
```

La note N1923 change ce comportement. Désormais, le qualificateur s'applique à la fois au tableau et à ses éléments:

```
const double a[2][2] = { ... }; // en C23: un tableau constant contenant des
const double.
```

Avec une exception: un tableau n'est jamais qualifié d'`_Atomic`, mais ses éléments peuvent l'être.

```
typedef int A[1][2];
const A a = {{4, 5}};  // en C23: un tableau const de tableau const
int* b = a[0];         // Erreur depuis C89 (b doit être const)
void *ptr = a;         // Erreur depuis C23 (ptr doit être const)
```

Changement sur les assertions

Il existe en C deux fonctions permettant de vérifier que des conditions sont bien vérifiées, et d'arrêter le programme dans le cas contraire. Il s'agit de `assert` et `static_assert`. Le premier vérifie des choses lors de l'exécution et le second lors de la compilation. Tous les deux ont reçu quelques changements.

Tout d'abord le static_assert. Normalement il s'utilise avec en paramètre une expression, et un message à afficher par le compilateur :

```
int tableau1[10000];
int tableau2[10000];

static_assert(sizeof(tableau1) == sizeof(tableau2), "Les deux tableaux
doivent avoir la même taille");
```

La même syntaxe est disponible en C++. Cependant, la dernière version de C++ a rendu optionnel le message, qui n'apporte finalement pas grand-chose la plupart du temps. C23 intègre également ce changement et rend le message optionnel. On pourra donc écrire plus simplement :

```
static_assert(sizeof(tableau1) == sizeof(tableau2));
```

Du côté de `assert()`, le problème est qu'il est défini sous forme d'une macro et que cette dernière manque de parenthésage pour protéger les paramètres.

Par exemple si on écrit:

```
    assert((int[2]){1,2}[0]);
```

Le préprocesseur va découper les arguments à assert en deux: `(int[2]){1` d'un côté et `2}[0]` de l'autre. Ce qui bien sûr se termine par une erreur du compilateur. La note N2829 corrige ce problème à la fois pour C et C++.

Changement sur les labels

En C++, les *labels* sont utilisés de deux façons :

Avec l'instruction `goto` :

```
void function(void)
{
    int* a = malloc(...);

    ...

    if (error)
        goto end; // continuer l'exécution au niveau du label "end"

    ...

end:              // (ici)
   free(a);
}
```

Et dans les `switch`:

```
switch(a) {
    case 1: // un autre type de label
        break;

    default:
        break;
}
```

Pour des raisons historiques, les cas suivants n'étaient pas autorisés :

```
void f(int x)
{
restart:                 // interdit: label pointant sur une déclaration de
variable
    int z = 3 * x;
    switch (x) {
        case 1:          // interdit: label pointant sur une déclaration de
variable
            int y = z;
            if (y == 3)
                goto out;
            if (y > z)
                goto restart;
            break;
        default:         // interdit: label à la fin d'un bloc
    }
out:                     // interdit: label à la fin d'un bloc
}
```

La note N2496 rend le code ci-dessus valide. Cependant, quelques cas posaient encore problème et la norme a été à nouveau modifiée par N2508 puis par N2663 pour les corriger.

Les types entiers à taille exacte peuvent être plus grands que uintmax_t

En C il existe différents types d'entier, en plus des classiques `int` et `unsigned int`, et de leurs variantes `long` et `short`, on trouve également des types à taille exacte, par exemple `int32_t` qui fait 32 bits. La norme définit également un `uintmax_t` qui doit être le plus grand type disponible. Par exemple, si tous les entiers peuvent être représentés sur 64 bits, alors `uintmax_t` devrait faire 64 bits.

Cela pose problème parce que de nouveaux types peuvent apparaître dans de nouvelles versions du compilateur. Par exemple, gcc peut compiler du code avec des entiers sur 128 bits, avec un type actuellement nommé `__int128_t`. Mais le type `intmax_t` a été défini pour être sur 64 bits, et il est utilisé à de nombreux endroits dans le code compilé par gcc. Changer la taille de `intmax_t` provoquerait donc un changement d'ABI et de nombreux problèmes de compatibilité.

La note N2888 autorise donc les compilateurs à déclarer des types `uintmax_t` et `intmax_t` qui ne sont en fait pas de la taille maximale. Permettant à gcc d'ajouter des entiers sur 128 bits avec le nom `int128_t` sans avoir besoin de changer `intmax_t`.

Il faudra rester prudent avec l'utilisation de ces types car `intmax_t` reste le type utilisé par le préprocesseur pour manipuler des entiers. Il y a donc un risque d'overflow si on utilise le préprocesseur avec des entiers de taille plus large (par exemple la constante `INT128_MAX` définie dans `stdint.h`).

Nouveautés dans le préprocesseur

et

Le préprocesseur C permet de faire de la compilation conditionnelle :

```
#if UNE_MACRO
// code compilé si UNE_MACRO ne vaut pas 0
#else
// code compilé dans les autres cas
#endif
```

On peut aussi tester si une macro est simplement définie (même si elle vaut 0)

```
#if defined(UNE_MACRO)
// code compilé si UNE_MACRO est définie
#else
// code compilé dans les autres cas
#endif
```

On peut chaîner plusieurs conditions avec `#elif`:

```
#if defined(LINUX)
// code spécifique à Linux
#elif defined(BSD)
```

```
// code spécifique à BSD
#else
// code pour les autres systèmes
#endif
```

Et on peut abréger `#if defined` en `#ifdef`:

```
#ifdef(LINUX)
// code spécifique à Linux
#elif defined(BSD)
// code spécifique à BSD
#else
// code pour les autres systèmes
#endif
```

Il manquait la possibilité d'abréger `#elif defined` en `#elifdef` et `#elif !defined` en `#elifndef`. Ces oublis sont corrigés par N2645

- Générer un avertissement pendant la compilation

Il existe en C une directive `#error`:

```
#if sizeof(size_t) < 8
#error "Ce programme a besoin d'un espace d'adressage d'au moins 64 bits"
#endif
```

La note N2686 ajoute la directive `#warning`:

```
#if sizeof(size_t) < 8
#warning "Ce programme n'a pas été testé sur un système 32 bits"
#endif
```

On peut supposer que `#error` fait échouer la compilation, cependant, ce n'est pas obligatoire dans la norme C qui ne s'intéresse pas du tout à la façon dont le code va être interprété ou compilé, et donc ne peut pas spécifier le résultat d'un éventuel compilateur.

La directive `#warning` était déjà disponible comme extension dans plusieurs compilateurs, désormais elle fait officiellement partie du langage C normalisé.

- Inclure des données binaires dans le fichier généré

La note N3017 ajoute une nouvelle directive #embed qui permet d'inclure des données binaires directement dans le programme compilé.

L'utilisation ressemble un peu à celle de `#include`, par exemple:

```
const unsigned char icon_display_data[] = {
     #embed "art.png"
};
```

Mais plein d'options sont disponibles, par exemple pour inclure seulement 512 octets provenant d'un fichier :

```
const int please_dont_oom_kill_me[] = {
    #embed "/dev/urandom" limit(512)
};
```

Les options possibles sont :

- limit : définit le nombre d'octets à inclure
- prefix et suffix : ajoutent des caractères avant et après le contenu inclus, seulement si celui-ci est non-vide
- if_empty pour ajouter des caractères si le fichier à inclure est vide

Il y a également un `__has_embed` qui fonctionne comme `__has_include` pour tester si le fichier à inclure est disponible.

Une utilisation amusante est pour écrire une *quine* (un programme qui affiche son propre code source):

```
#include <stdio.h>

int main(void)
{
    const char* source = {
        #embed __FILE__ suffix(, 'f0c5d44e332b68159a38346b2638e012bbefdf3f')
// On ajoute un f0c5d44e332b68159a38346b2638e012bbefdf3f à la fin pour
pouvoir utiliser la chaîne de caractères avec puts
    ;}
    puts(source);
    return 0;
}
```

Pragmas pour contrôler les arrondis sur les calculs en virgule flottante

```
float default_mode = 1.0f / 3.0f; // valeur par défaut de l'arrondi, non
définie dans la norme

#pragma STDC FENV_ROUND FE_UPWARD
float up = 1.0f / 3.0f; // arrondi par au-dessus (le résultat sera
0.33....34)

#pragma STDC FENV_ROUND FE_DOWNWARD
float down = 1.0f / 3.0f; // arrondi par au-dessous (le résultat sera
0.33....33)

#pragma STDC FENV_ROUND FE_DYNAMIC
float dynamic = 1.0f / 3.0f; // arrondi au plus proche
```

Le comportement des arrondis pouvait déjà être modifié avec la fonction `fesetround`, mais cela demande pas mal de code pour sauvegarder et restaurer l'environnement (avec `fegetenv` et `fesetenv`) autour de chaque opération qui a besoin d'un mode spécifique.

L'utilisation de pragmas permet au compilateur de savoir quelles parties du code utilisent quel type d'arrondi, et d'optimiser les sauvegardes et changements de modes pour ne conserver que ceux qui sont nécessaires.

Les nouvelles fonctionnalités de la libc

Fonctions mathématiques protégées contre les débordements

En C, les entiers peuvent déborder, par exemple :

```
uint32_t i = UINT32_MAX; // la plus grande valeur possible pour un entier
non-signé sur 32 bits
i = i + 1; // maintenant i vaut 0
```

C'est utile dans certains cas, mais la plupart du temps, c'est une source de problèmes. La note N2683 introduit des fonctions permettant de faire des calculs avec vérification de débordement:

```
#include <stdckdint.h> // nouvel en-tête ajouté par cette note

uint32_t i = UINT32_MAX;
if (ckd_add(&i, 1, i) == true) {
    // débordement détecté
}
```

La note propose d'autres extensions (avec de nouveaux types pour rendre cela plus transparent à l'utilisation) mais pour l'instant ces idées n'ont pas été intégrées dans le standard. Peut-être dans une prochaine version.

Préservation des dans les fonctions standard

Ce changement concerne les fonctions suivantes: `bsearch`, `bsearch_s`, `memchr`, `strchr`, `strpbrk`, `strrchr`, `strstr`, `wcschr`, `wcspbrk`, `wcsrchr`, `wcsstr`, et `wmemchr`.

Les valeurs de retour de ces fonctions ne sont pas qualifiées `const`. Par exemple pour strchr:

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

Ceci permet d'utiliser ces fonctions dans deux cas: avec un paramètre un buffer constant, ou un buffer qui va ensuite être modifié.

```
const char* const uneConstante = "Du texte non modifiable";
char* const uneVariable = strdup(uneConstante); // cette chaîne peut être
modifiée
```

```
// Recherche dans une chaîne constante, résultat constant:
const char* const pointeur = strchr(uneConstante, 'n');

// Recherche dans une chaîne non constante, le pointeur retourné peut être
utilisé pour modifier la chaîne:
char* const autrePointeur = strchr(uneVariable, 'n');
*autrePointeur = 'b';

// Rien n'empêche d'écrire ceci :
char* const dernierPointeur = strchr(uneConstante, 'n');
*dernierPointeur = 'b'; // comportement indéfini: écriture dans de la
mémoire constante

free(uneVariable);
```

Le problème est corrigé depuis longtemps en C++ qui propose simplement deux prototypes pour les fonctions concernées :

```
// Si le buffer en entrée est constant, alors la valeur de retour aussi:
const char *strchr(const char *s, int c);

// Sinon, les deux sont modifiables:
char *strchr(char *s, int c);
```

Le problème est que cette solution exploite la possibilité en C++ d'avoir plusieurs fonctions avec le même nom mais des paramètres de types différents. Ce n'est pas possible en C.

La note N3020 propose donc une solution différente avec le même résultat. La note suggère une implémentation possible:

```
#define IS_POINTER_CONST(P) _Generic(1 ? (P) : (void *)(P) \
 , void const *: 1 \
 , default : 0)
#define STATIC_IF(P, T, E) _Generic (&(char [!!(P) + 1]) {0} \
 , char (*) [2] : T \
 , char (*) [1] : E)
#define _STRING_SEARCH_QP(T, F, S, ...) \
 STATIC_IF (IS_POINTER_CONST ((S)) \
 , (T const *) (F) ((S), __VA_ARGS__) \
 , (T *) (F) ((S), __VA_ARGS__))

#define strstr(S1, S2) _STRING_SEARCH_QP(char, strstr, (S1), (S2))
```

Cela conduit à quelques acrobaties normatives pour spécifier exactement le comportement attendu, sans imposer une méthode d'implémentation. Et il reste un cas de comportement indéfini si l'argument passé à l'une de ces fonctions est un pointeur NULL (le comportement de la macro ci-dessus est d'utiliser l'implémentation non-const de la fonction, contrairement au C++ ou on obtient normalement une erreur de compilation).

Fonctions POSIX

Les fonctions suivantes font leur apparition. Elles étaient déjà présentes dans la spécification POSIX pour pallier certains problèmes des fonctions standards du langage C, mais toutes les implémentations du C ne ciblent pas des systèmes compatibles avec POSIX.

- `memccpy()` : copie mémoire d'une zone mémoire vers une autre, pour un nombre d'octets maximal, ou jusqu'à rencontrer un certain caractère (par exemple `\0`).
- `strdup()` : copie une chaîne de caractères (terminée par `null`) dans un espace mémoire alloué automatiquement (et qui doit être libéré en utilisant `free`).
- `strndup()` : comme `strdup`, en rajoutant une contrainte de taille maximale.

Ajout de fonctions réentrantes pour le formatage de date, avec un suffixe `_r`. Elles sont équivalentes aux fonctions sans suffixe, mais utilisent un espace mémoire passé en paramètre à la place d'un espace global unique. Voir : `asctime_r()`, `ctime_r()`, `gmtime_r()`, `localtime_r()`.

```
void compare_year(time_t start, time_t end)
{
    struct tm* start_tm = gmtime(start);
    struct tm* end_tm = gmtime(end); // Problème: le deuxième appel à gmtime
va écraser le résultat du premier!

    if (start_tm->tm_year != end->tm_year) {
        printf("L'année a changé");
    }
}

void compare_year_c23(time_t start, time_t end)
{
    struct tm start_tm;
    gmtime_r(start , &start_tm);
    struct tm end_tm;
    gmtime_r(end, &end_tm); // Pas de problème, les deux struct tm sont bien
distinctes

    if (start_tm.tm_year != end.tm_year) {
        printf("L'année a changé");
    }
}
```

Format alternatif pour les mois dans strftime

Les fonctions `strftime()` et `wcsftime()` permettent en C23 d'utiliser les formats %0b et %0B pour le formatage des mois (en version abrégée et en version longue).

Ceci permet par exemple de formater la date correctement dans tous les cas en russe et en ukrainien, dont la grammaire a besoin de deux formes (nominatif et génitif) pour les noms de mois en fonction du contexte.

Dix années d'efforts des développeurs de la glibc pour modifier la grammaire de toutes les langues

concernées ayant échoué, la complexité de ces langues est donc prise en compte (de façon limitée) avec cette solution. Le préfixe O était déjà standardisé pour l'utilisation dans d'autre cas dans le formatage de dates, mais pas pour les noms de mois.

timespec_getres()

En plus des changements listés ci-dessus sur les fonctions existantes de gestion du temps, la note N2417 ajoute une nouvelle fonction `timespec_getres()`, qui permet de connaître la résolution (précision) des valeurs retournées par `timespec_get()`.

Cette note apporte également de nombreux changements sur d'autres fonctions liées à la gestion du temps: modification des prototypes de ces fonctions pour interdire le passage accidentel d'un pointeur nul, dépréciation de la fonction historique `clock`, et diverses clarifications dans la spécification.

Extensions pour la famille des fonctions et

La version C99 de la norme avait ajouté des types entiers avec des tailles spécifiques. Historiquement, le C proposait les types `int`, `short` et `long` dont le nombre de bits est déterminé en fonction de ce qui est raisonnable sur l'architecture matérielle ciblée.

En C99, des types comme `uint8_t` sont apparus (uint8_t est un entier non-signé occupant exactement 8 bits). Ces types sont optionnels et ne sont présents que si l'architecture matérielle permet de les implémenter efficacement.

L'utilisation de ces types avec les fonctions de formatage (`printf` ou `scanf` par exemple) posait problème parce que ces fonctions ont besoin d'une chaîne de caractères décrivant le type à formater. Par exemple :

```
int unEntier = 0;
printf("un entier: %d\n", unEntier);

unsigned unEntierNonSigne = 0;
printf("un entier non signé: %u\n", unEntierNonSigne);

long unGrosEntier = 0;
printf("un gros entier: %ld\n", unGrosEntier);
```

La solution retenue pour les types à taille fixe était d'utiliser des macros, remplacées par le compilateur par la bonne chaîne de formatage :

```
int8_t unEntier8Bits;
printf("un entier sur 8 bits: %" PRId8 "\n", unEntier8Bits); // la macro
PRId8 est remplacée par "d" par exemple

int32_t unEntier32Bits;
printf("un entier sur 32 bits: %" PRId32 "\n", unEntier32Bits); // la macro
PRId32 est remplacée par "d" ou "ld" selon que int32 est en fait un int ou
un long int
```

Cette solution est peu pratique à utiliser. La note N2680 introduit un nouveau système de formatage

et on pourra désormais écrire:

```
int8_t unEntier8Bits;
printf("un entier sur 8 bits: %w8d\n", unEntier8Bits);

int32_t unEntier32Bits;
printf("un entier sur 32 bits: %w32d\n", unEntier32Bits);
```

Cela permettra de corriger quelques surprises lors de l'utilisation des macros précédentes :

```
printf("un entier sur 8 bits: %" PRId8 "\n", 0xFF); // affiche 255 à cause
des règles de promotion de type
printf("un entier sur 8 bits: %w8d\n", 0xFF); // affiche -1
```

Le préfixe wf (au lieu de seulement w) est également disponible pour la famille de types `int_fastN_t`.

Macro constantes indiquant la largeur des types entiers

Ces constantes sont ajoutées par la note N2412 suite à la normalisation de l'utilisation du complément à 2 pour représenter les entiers signés (auparavant, la norme ne spécifiait pas de représentation spécifique et ne pouvait donc pas dire grand-chose de la représentation interne des nombres).

Il s'agit des constantes `INT8_WIDTH`, `INT16_WIDTH`, `INT32_WIDTH`, `INT64_WIDTH` de leurs équivalents `INT_FASTN_WIDTH` et `INT_LEAST8_WIDTH` pour les types int_fastN_t et int_leastN_t, ainsi que `INTPTR_WIDTH` et `INTMAX_WIDTH`, indiquant chacune le nombre de bits dans le type concerné. Elles rejoignent CHAR_WIDTH qui existait déjà dans les versions précédentes du C.

Nettoyage de données sensibles

Les données sensibles telles que les mots de passe, devraient être nettoyées de la mémoire dès qu'elles ne sont plus nécessaires.

La fonction `memset_explicit()` a été normalisée afin de pouvoir exprimer ce besoin (N2897).

Ainsi le compilateur ne risque plus d'optimiser cette portion de code et de mettre de côté le nettoyage de la zone mémoire concernée.

```
void erase_password(char* password, size_t size)
{
    memset_explicit(password, 0, size); // l'utilisation d'un memset simple
ici pourrait être optimisée par le compilateur : l'espace mémoire est
libérée juste après, donc toute écriture dedans est inutile du point de vue
du fonctionnement du programme.
    free(password);
}
```

La très similaire fonction `memset_s` avait déjà été ajoutée dans la norme C11, mais dans l'annexe K, dont l'implémentation n'est pas obligatoire. La note N1967 fait un retour d'expérience sur l'utilisation

des fonctions de l’annexe K.

Macro pour tester les fonctionnalités

Depuis C95, la macro __STDC_VERSION__ permet de savoir, à la précompilation, sur quelle version de la norme le code peut se reposer.

La macro est étendue avec l’année de parution et la révision de la norme en décimal dans un `long`.

```
#if defined(__STDC_VERSION__) && __STDC_VERSION__ >= 199409L
puts("Le compilateur implémente C95 ou une version plus récente.");
#elif defined(__STDC__)
puts("Le compilateur implémente C89.");
#endif
```

C23 étend ce concept à certaines bibliothèques standards. Elles peuvent ainsi avoir leur propre cycle de vie. Par exemple les constantes suivantes sont définies avec l’identifiant de la norme :

- __STDC_VERSION_FENV_H__
- __STDC_VERSION_MATH_H__
- __STDC_VERSION_STDINT_H__
- __STDC_VERSION_STDLIB_H__
- __STDC_VERSION_TGMATH_H__
- __STDC_VERSION_TIME_H__

Ces fichiers d’en-tête sont en général fournis par la bibliothèque C et ne sont pas forcément synchronisés avec le compilateur. Il est donc, en théorie, possible d’obtenir un système ou la version du C utilisée par le compilateur n’est pas la même que celle implémentée par la bibliothèque standard.

Ces macros permettront donc de vérifier plus finement la version de chacun des composants.

Cela permettra également de publier des spécifications pour le langage C découpées en plusieurs standards séparés, pour mettre à jour seulement un ou quelques-uns de ces fichiers sans modifier la version de tout le langage.

Conclusion

Cette version du C apporte beaucoup de changements et permet de resynchroniser un certain nombre de fonctionnalités avec des évolutions présentes depuis déjà quelques années dans C++.

Il faut maintenant espérer que ces évolutions seront rapidement intégrées dans les compilateurs et utilisées par les développeurs. Historiquement, le déploiement des nouvelles versions du C est relativement lent. Aujourd’hui il est encore assez courant de trouver des projets développés en C99 ou même parfois en C89, alors que deux versions plus récentes du langage sont déjà disponibles depuis plusieurs années.

Exercices d'application :

-1- Écrivez un programme qui :

1. affiche « Salut toi, appuie sur une touche s'il te plaît » ;
2. attend l'appui d'une touche ;
3. affiche : « Merci d'avoir appuyé sur une touche ».

Une fois que vous serez satisfait de votre solution, vous pourrez la comparer avec la solution qui apparaît un peu plus loin.

Sous Linux, il est possible d'éviter de retaper à chaque fois les commandes : Pour cela, il suffit d'appuyer plusieurs fois sur la flèche vers le haut Image non disponible, ce qui fera réapparaître les dernières commandes validées. Les flèches haut Image non disponible et bas Image non disponible permettent ainsi de circuler dans l'historique des commandes entrées.

-1- Corrigé de l'exercice du chapitre :

```c
#include <stdio.h>
int main () {
   /* Affiche premier message */
   puts ("Salut toi, appuie sur une touche s'il te plaît");
   getchar (); /* Attend la frappe d'une touche */
   /* Affiche le second message */
   puts ("Merci d'avoir appuyé sur une touche");
   return 0;
}
```

- Autres exercices_en_langage_C