

# Les nouveautés du C99

par Nicolas Joseph ([home](#)) ([Blog](#))

Date de publication : 27 Avril 2006

Dernière mise à jour : 16 Décembre 2008

Au travers de cet article, je vous propose un tour d'horizon des nouveautés du C99.

|                                                             |    |
|-------------------------------------------------------------|----|
| Historique.....                                             | 3  |
| I - Introduction.....                                       | 4  |
| II - Préprocesseur.....                                     | 5  |
| II-A - La macro <code>__STDC_VERSION__</code> .....         | 5  |
| II-B - Macros à nombre variable d'arguments.....            | 5  |
| II-C - Les commentaires mono-ligne.....                     | 5  |
| II-D - La directive <code>#line</code> .....                | 5  |
| II-E - La directive <code>#pragma</code> .....              | 5  |
| III - Syntaxe.....                                          | 6  |
| III-A - Les nouveaux mots réservés.....                     | 6  |
| III-B - Séquences d'échappement à deux caractères.....      | 6  |
| III-C - Mélange déclaration/code.....                       | 6  |
| III-D - Initialisation des membres d'une structure.....     | 6  |
| IV - Sémantique.....                                        | 8  |
| IV-A - Les fonctions en ligne.....                          | 8  |
| IV-B - <code>long long</code> .....                         | 8  |
| IV-C - <code>__func__</code> .....                          | 8  |
| IV-D - Type de retour implicite.....                        | 8  |
| IV-E - Les pointeurs restreints.....                        | 9  |
| IV-F - Les tableaux de taille variable.....                 | 9  |
| IV-G - Type de tableau incomplet.....                       | 10 |
| V - La bibliothèque standard.....                           | 12 |
| V-A - Les nouveaux fichiers d'en-tête.....                  | 12 |
| V-B - Les types entiers étendus.....                        | 12 |
| V-C - Les caractères étendus.....                           | 12 |
| V-D - <code>va_copy</code> .....                            | 12 |
| V-E - Le type booléen.....                                  | 13 |
| V-F - Les nombres complexes.....                            | 13 |
| V-G - Les nombres à virgules flottantes.....                | 13 |
| V-G-1 - Dépassement de capacité de la valeur de retour..... | 13 |
| V-G-2 - Le controle des arrondis.....                       | 14 |
| V-G-3 - Les exceptions.....                                 | 14 |
| V-G-4 - Manipulation de l'environnement.....                | 15 |
| VI - Conclusion.....                                        | 16 |
| VII - Références.....                                       | 17 |
| VIII - Remerciements.....                                   | 18 |

## Historique

### 4 Novembre 2006

- Clarification de la signification des pragma
- Précision sur l'environnement en virgule flottante
- Ajout des nouveaux fichiers d'en-tête
- Ajout de références

## I - Introduction

A l'heure actuelle, deux normes cohabitent : il y a celle publiée en 1989 par l'ANSI (1), reprise en 1990 par l'ISO (2) connue respectivement sous le nom de C89 et C90 (ce sont les mêmes) et une seconde norme, qui regroupe les différents *drafts* apparues depuis la dernière norme, publiée en 1999 par ISO connue sous le nom de C99 (et C2k pour la version ANSI). Cette dernière a pour but de minimiser les incompatibilités entre le C et le C++ (sans pour autant en faire un seul et même langage). Voici donc la liste des nouveautés et des modifications apportées par la norme C99 par rapport au C90.

## II - Préprocesseur

### II-A - La macro `__STDC_VERSION__`

Cette macro permet de connaître la norme utilisée :

- En C89 elle n'est pas définie
- En C94 elle vaut 199409L
- Et maintenant, en C99, 199901L.

### II-B - Macros à nombre variable d'arguments

Cette fonctionnalité permet de définir des macros dont le nombre des arguments peut être différent d'un appel à l'autre. La notation utilisée est la même que pour les fonctions ('...') et la liste est récupérée grâce à la macro `__VA_ARGS__` :

```
#define print_error(s, ...) fprintf (stderr, (s), __VA_ARGS__)
```

### II-C - Les commentaires mono-ligne

Les commentaires "à la C++", c'est à dire commençant par `//` et se terminant à la fin de la ligne viennent s'ajouter aux commentaires classiques (`/* */`).

### II-D - La directive `#line`

Cette directive du préprocesseur permet de modifier le numéro de la ligne courante (et optionnellement le nom du fichier). La valeur utilisée était limitée à  $2^{15}-1$  en C90 et passe maintenant à  $2^{31}-1$ .

### II-E - La directive `#pragma`

La directive **#pragma** permet de contrôler le comportement du compilateur. La norme C99 définit trois pragma standards :

- **FP\_CONTRACT** : optimisation des expressions flottantes
- **FENV\_ACCESS** : notre programme accède aux paramètres du FPU (*Floating Point Unit* ou unité de calcul flottant), ceci se fait à l'aide des fonctions déclarées dans *fenv.h*. Le compilateur doit donc respecter la norme pour les opérations sur les nombres flottant
- **CX\_LIMITED\_RANGE** : dans le cas d'opérations sûres avec les nombres complexes (c'est à dire sans risque de débordement de capacités), vous pouvez le signaler afin d'accélérer le traitement en activant ce pragma.

Chacune de ces fonctionnalités peuvent être activée (**ON**), désactivée (**OFF**) ou remis à son état par défaut (**DEFAULT**). Par exemple, voici comment activer les optimisations des expressions flottantes :

```
#pragma STDC FP_CONTRACT ON
```



*Chaque compilateur propose ces propres directives pragma. Si le compilateur rencontre une directive inconnue, la norme prévoit que celle-ci soit ignorée, cependant il est déconseillé d'utiliser des directives non standard car selon les compilateurs leur rôle peut être différent.*



*A la place de la directive **#pragma**, il est possible d'utiliser l'opérateur unaire **\_Pragma**.*

## III - Syntaxe

### III-A - Les nouveaux mots réservés

Le C99 ajoute cinq mots réservés au langage :

- **restrict**
- **inline**
- **\_Complex**
- **\_Imaginary**
- **\_Bool**

### III-B - Séquences d'échappement à deux caractères

Les séquences d'échappement à deux caractères permettent de remplacer certains caractères spéciaux qui peuvent être absent du clavier (comme les séquences d'échappement à trois caractères). Avant tout traitement, les lexèmes <:, >:, <%, %>, %: et %::%: sont respectivement remplacés par [, ], {, }, # et ##.

### III-C - Mélange déclaration/code

En C90, les déclarations doivent être faites au début d'un bloc, alors qu'en C99 il est tout à fait possible d'écrire :

```
int main (void)
{
    printf ("Debut du code\n");
    int i = 10;
    printf ("i vaut %d\n", i);
    return 0;
}
```

Cependant ce genre de mélange n'est pas recommandé afin de préserver la lisibilité du code.

Cette fonctionnalité entraîne la possibilité de créer une variable de boucle temporaire pour l'instruction **for** :

```
for (int i = 0; i < 10; i++)
{
    /* variable i utilisable */
}
/* variable i inexistante */
```

Est équivalent à :

```
{
    int i;

    for (i = 0; i < 10; i++)
    {
        /* variable i utilisable */
    }
}
/* variable i inexistante */
```

Par conséquent, la variable *i* n'existera plus à la sortie de la boucle.

### III-D - Initialisation des membres d'une structure

Il est possible d'initialiser les membres d'une structure, lors de sa déclaration, en utilisant leurs noms :

```
struct T
{
    int a;
    int b;
};

struct T s1 = {.a = 0, .b = 1};
/* ou dans le cas d'un tableau de structure : */
struct T s2[] = {[0].a = 0, [0].b = 0, [1].a = 1, [1].b = 1};
```

De plus, pour les structures de type **auto**, il est possible d'initialiser les membres avec des valeurs non connues au moment de la compilation :

```
int i = 0;
struct T s1 = {.a = i, .b = i+2};
```

## IV - Sémantique

### IV-A - Les fonctions en ligne

Le mot-clé **inline** est un nouvel attribut pour les fonctions qui ont pour but de remplacer le système de macro défini grâce à la commande du préprocesseur **#define**. Pour illustrer l'avantage de cette méthode par rapport aux macros, voici l'exemple classique du calcul du carré d'un nombre :

```
#define carre(a) ((a)*(a))
```

Si par malheur vous oubliez les parenthèses autour de la variable **a**, la macro peut produire un résultat inattendu, il faut aussi faire attention au effets de bord puisque la variable **a** est évaluée deux fois. Maintenant avec le mécanisme de fonction en ligne :

```
inline int carre (int a)
{
    return a*a;
}

int main (void)
{
    int b = carre (5);

    printf ("Le carre de 5 est %d\n", b);
    return 0;
}
```

Ceci supprime les dangers liés à l'utilisation de l'instruction **#define**. Les fonctions en ligne doivent être définies aux mêmes endroits que les macros classiques, ce qui peut alourdir les fichiers d'en-tête et surtout recopie la fonction dans plusieurs fichiers sources : pour éviter de se retrouver avec plusieurs fonctions globales portant le même nom, il est conseillé de toujours utiliser le mot-clé **inline** conjointement avec le mot-clé **static** (même dans un fichier d'en-tête).

 L'attribut **inline** est une information d'optimisation pour le compilateur, si la mise en ligne n'est pas possible, le compilateur peut ignorer cet attribut.

### IV-B - long long

Le nouveau type d'entier **long long** (signé ou non) est représenté sur au moins 64 bits. Il s'accompagne des notation **LL** (ou **ll**) et **ULL** (ou **ull**).

### IV-C - \_\_func\_\_

La variable **\_\_func\_\_** (3) contient le nom de la fonction courante. Elle est déclarée ainsi :

```
static const char __func__[];
```

### IV-D - Type de retour implicite

En C90, en l'absence de type de retour, le compilateur considère que cette fonction retourne un **int**, en C99 ce n'est plus possible : le type de retour doit être spécifié.



## IV-E - Les pointeurs restreints

Le mot-clé **restrict** permet de spécifier qu'un pointeur est le seul à pointer sur une zone mémoire. Cela permet au compilateur d'opérer certaines optimisations. Par exemple, la fonction *memcpy* qui est utilisée pour copier deux zones de mémoires qui ne se chevauchent pas peut avoir comme prototype, en C99 :

```
void *memcpy (void *restrict dst, const void *restrict src, size_t size);
```

Le mot-clé se place entre le caractère \* et le nom du pointeur.

 Comme pour les fonctions en ligne, cette directive peut être ignorée par le compilateur.

## IV-F - Les tableaux de taille variable

Voilà une fonctionnalité qui va réjouir les réfractaires à l'utilisation des fonctions d'allocation dynamique. En effet, en C90 pour déclarer un tableau, sa taille doit être connue à la compilation. Le C99 propose un mécanisme nommé VLA (**V**ariable-**L**ength **A**rrays), qui permet de créer des tableaux dont la taille n'est connue qu'à l'exécution. Voici un exemple :

```
void foo (int n)
{
    int tab[n];
    ...
}
```

Comme toutes variables automatiques, elles sont détruites à la fin du bloc dans lequel elles sont déclarées (plus de fuites mémoires avec ce système), en contre partie l'allocation dynamique reste indispensable pour retourner l'adresse d'une zone mémoire.

On peut aussi déclarer un tableau de taille variable dans le prototype d'une fonction :

```
void foo (int n, int a[n]) { ... }
```

Il faut que la variable *n* soit définie avant, ce qui interdit d'écrire ceci :

```
void foo (int a[n], int n) { ... }
```

La déclaration d'une telle fonction peut s'écrire :

```
/* Declaration complete */
void foo (int n, int a[n]);
/* Declaration partielle */
void foo (int n, int a[*]);
```

 Notez l'utilisation du caractère \*\* pour signaler un tableau de taille variable.

Malgré le warning de gcc, le code fonctionne comme prévu.

Ce mécanisme très puissant a tout de même certaines limitations :

- Les tableaux de tailles variables sont limités au fichier, il n'est pas possible d'utiliser le mot-clé **extern** :

```
extern int n;
int a[n]; /* n est declaree extern : utilisation interdite */
extern int b[n]; /* utilisation interdite */
```

- Il n'est pas possible de créer de tel tableau **static**. Par contre il est possible de déclarer **static** un pointeur sur VLA :

```
int b[10];

int foo (int n)
{
    static int a[n]; /* Interdit */
    static (*p_b)[n] = &b; /* Autorise si n == 10 */
}
```



*Déclarer un pointeur sur un VLA d'une taille différente de celle autorisée ne stoppe pas l'exécution du programme (on pourrait s'attendre à la levée d'une exception ou quelque chose approchant), mais bien sur ce genre de manipulation est très risquée et est à classer dans les comportements indéfinis.*

- Il n'est pas possible de déclarer un tableau de taille variable dans une structure ou une union
- Il n'est pas possible d'effectuer un saut dans un bloc qui contient une déclaration de ce type de tableau, par contre sortir de ce bloque est autorisé :

```
goto start; /* Interdit */
{
    int n = 10;
    int a[n];

start :
    /* ... */
    goto end; /* Autorise */
}
end :
    /* ... */
```

## IV-G - Type de tableau incomplet

Lors de la déclaration d'une structure, il est possible de ne pas spécifier la taille d'un tableau, s'il s'agit du dernier membre de la structure :

```
struct incomplet
{
    int n;
    int tab[];
};
```

Cela permet de faire de belles choses comme ceci :

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

struct incomplet
{
    int n;
    int tab[];
};

int main (void)
{
    struct incomplet *a = NULL;

    a = malloc (sizeof (*a) + 3 * sizeof (int));
    if (a)
    {
        a->n = 3;
        for (int i = 0; i < a->n; i++)
        {
            a->tab[i] = i;
        }
        printf ("a->tab = { ");
    }
```

```
    for (int i = 0; i < a->n; i++)
    {
        printf ("%d, ", a->tab[i]);
    }
    printf (" }; \n");
    free (a), a = NULL;
}

struct incomplet *b = NULL;

b = malloc (sizeof (*b) + 5 * sizeof (int));
if (b)
{
    b->n = 5;
    for (int i = 0; i < b->n; i++)
    {
        b->tab[i] = i;
    }
    printf ("b->tab = { ");
    for (int i = 0; i < b->n; i++)
    {
        printf ("%d, ", b->tab[i]);
    }
    printf (" }; \n");
    free (b), b = NULL;
}
return 0;
}
```

```
gege2061@debian:~/c99$ gcc -Wall -std=c99 main.c && ./a.out
a->tab = { 0, 1, 2, };
b->tab = { 0, 1, 2, 3, 4, };
```

Cela évite d'utiliser un pointeur pour *tab* et d'avoir une allocation supplémentaire à faire. Hélas il n'est pas possible de faire :

```
struct incomplet a = {3, 0, 1, 2};
struct incomplet b = {5, 0, 1, 2, 3, 4};
```

## V - La bibliothèque standard

### V-A - Les nouveaux fichiers d'en-tête

Un enrichissement de la bibliothèque standard, s'accompagne logiquement de nouveaux fichiers d'en-tête :

- `complex.h`
- `fenv.h`
- `inttypes.h`
- `iso646.h`
- `stdbool.h`
- `stdint.h`
- `tgmath.h`
- `wchar.h`
- `wctype.h`

### V-B - Les types entiers étendus

En C, la taille des types de variables n'est pas définie par la norme, seul une taille minimale est garantie. Le C99, dans le fichier d'en-tête `stdint.h` propose des types dont la taille est connue :

| Types                      | Description                                                 |
|----------------------------|-------------------------------------------------------------|
| <code>intN_t</code>        | Entier signé de N (8, 16, 32 ou 64 (4) ) bits               |
| <code>uintN_t</code>       | Entier non signé de N (8, 16, 32 ou 64) bits                |
| <code>intmax_t</code>      | Entier signé de taille maximale                             |
| <code>uintmax_t</code>     | Entier non signé de taille maximale                         |
| <code>int_leastN_t</code>  | Entier signé d'au moins N (8, 16, 32 ou 64)bits             |
| <code>uint_leastN_t</code> | Entier non signé d'au moins N (8, 16, 32 ou 64) bits        |
| <code>int_fastN_t</code>   | Entier rapide (5) signé d'au moins N (8, 16, 32 ou 64) bits |
| <code>uint_fastN_t</code>  | Entier rapide non signé d'au moins N (8, 16, 32 ou 64) bits |

 Ces types sont notés comme optionnels dans la norme, par conséquent leur portabilité est réduite.

### V-C - Les caractères étendus

En C90 il n'était pas possible d'utiliser de caractères spécifiques d'une langue (par exemple, les caractères accentués pour le français) et ce, même dans les commentaires. Pour pallier ce problème, la norme C99 propose un nouveau type de caractères étendus nommés **wchar\_t** (pour *wide char*) défini dans `wchar.h`. Pour convertir un caractère étendu en entier, le type **int** peut ne pas être suffisant, à la place, il faut utiliser le type **wint\_t**.

Toutes les fonctions de manipulation des chaînes de caractères (de la forme `str*`) sont reprises sous la forme `wcs*` (par exemple `wcslen`, `wscmp` ou encore `wscpy`) ainsi que les fonctions `wprintf` et `wscanf` pour l'affichage et la saisie formatée de chaîne de caractères étendues.

### V-D - va\_copy

Le fichier d'en-tête `stdarg.h` contient une nouvelle fonction `va_copy` qui permet de copier un objet de type `va_list`.

## V-E - Le type booléen

Un booléen est un nouveau type de variable qui peut prendre uniquement la valeur vrai ou faux. La définition du type `bool` est faite dans le fichier d'en-tête `stdbool.h` et contient les définitions suivantes :

- **bool** : le type booléen
- **true** : la valeur vrai
- **false** : la valeur faux
- **\_\_bool\_true\_false\_are\_defined** : si cette macro est définie, les valeurs **bool**, **true** et **false** le sont aussi.

 La définition du type **bool** a été placée dans un fichier d'en-tête pour garantir la compatibilité avec les anciens codes (il n'est pas rare de voir un code contenant déjà une définition de `bool`). Si, pour une raison quelconque, vous ne souhaitez pas inclure `stdbool.h`, le langage contient le type **\_Bool** qui peut prendre les valeurs 0 (faux) et 1 (vrai).

## V-F - Les nombres complexes

Les nombres complexes font leurs entrées dans le langage C grâce au type **complex** défini dans `complex.h`. Comme il existe trois représentations pour les nombres réels, il existe trois types de nombres complexes :

```
float complex
double complex
long double complex
```

Pour construire un nombre complexe, il suffit d'additionner sa partie réelle et sa partie imaginaire :

```
double partie_reelle = 2.0;
double partie_imaginaire = 3.0;
double complex z = partie_reelle + partie_imaginaire * I;
```

La macro `I` permet de construire la partie imaginaire.

Pour retrouver les parties réelles et imaginaires d'un complexe, on peut utiliser, respectivement les fonctions `creal` et `cimag` définies pour les trois type de complexes :

```
float crealf (float complex z);
double creal (double complex z);
long double creall (long double complex z);

float cimagf (float complex z);
double cimag (double complex z);
long double cimagl (long double complex z);
```

La bibliothèque standard du C99 propose aussi des fonctions pour calculer le module (`cabs*`), l'argument (`carg*`) et la représentation selon la projection sur la sphère de Riemann (`cproj*`) d'un nombre complexe.

## V-G - Les nombres à virgules flottantes

C'est sûrement dans ce domaine que les nouveautés sont les plus nombreuses, puisque le C99 introduit la notion d'*environnement virgule flottante* qui permet une gestion améliorée des erreurs du système de calcul en nombres flottants.

### V-G-1 - Dépassement de capacité de la valeur de retour

En plus de la valeur `HUGE_VAL`, retournée lorsque le type de retour n'a pas la capacité nécessaire pour contenir le résultat, la norme C99 ajoute les valeurs `HUGE_VALF` (de type `float`) et `HUGE_VALL` (de type `long double`).

## V-G-2 - Le controle des arrondis

Le type d'arrondi effectué sur les nombres à virgules flottantes est représenté par les constantes définies dans *fenv.h* :

- FE\_TONEARREST : au plus proche (valeur par défaut)
- FE\_UPWARD : vers le haut
- FE\_DOWNWARD : vers le bas
- FE\_TOWARDZERO : vers zéro (vers le bas pour les nombres positifs, vers le haut pour les nombres négatifs).

Il est possible de connaître le mode utilisé grâce à la fonction :

```
int fegetround (void);
```

Et de le modifier grâce à :

```
int fesetround (int mode);
```

En cas de succès, cette fonction retourne 0.

## V-G-3 - Les exceptions

En cas d'erreur lors d'une opération sur les nombres en virgules flottantes, une exception est levée, elle peut être traitée par le système (par exemple en envoyant un signal SIGFPE pour les systèmes POSIX). Cependant, un signal n'est pas toujours envoyé, dans ce cas c'est au programmeur de les tester.

Il existe cinq exceptions, représentées par des macros :

- FE\_DIVBYZERO : division par zéro
- FE\_OVERFLOW : valeur trop grande
- FE\_UNDERFLOW : valeur trop petite
- FE\_INEXACT : résultat inexacte
- FE\_INVALID : opération indéfinie
- FE\_ALL\_EXCEPT : représente l'ensemble des exceptions.

Vous pouvez tester la levée d'une exception avec la fonction :

```
int fetestexcept (int exceptions);
```

Où *exceptions* peut être une combinaison des macros précédentes (grâce à un *ou binaire* |) et la fonction retourne les exceptions réellement levées parmi celles passées en argument (ou zéro si aucune n'est levée).

Vous pouvez lever une ou plusieurs exceptions grâce à :

```
void feraiseexcept (int exceptions);
```

Et inversement, baisser les drapeaux d'exceptions avec :

```
void feclearexcept (int exceptions);
```

Et enfin, vous pouvez sauvegarder et restaurer l'état d'un ou plusieurs drapeaux :

```
void fegetexceptflag (fexcept_t *svg, int exceptions);  
void fesetexceptflag (const fexcept_t *svg, int exceptions);
```

## V-G-4 - Manipulation de l'environnement

Vous pouvez enregistrer et sauvegarder la configuration de l'environnement avec la fonction :

```
void fegetenv (fenv_t *ecf);  
void fesetenv (const fenv_t *ecf);
```

La fonction *fesetenv* peut prendre en argument la valeur *FE\_DFL\_ENV* qui permet de restaurer l'état initial de l'environnement.

```
void feupdateenv (const fenv_t *ecf);
```

Cette fonction restaure l'environnement en préservant les drapeaux d'exception.

```
void feholdexcept (fenv_t *ecf);
```

*feholdexcept* désactive les exceptions en sauvegardant les paramètres actuels.

## VI - Conclusion

Voilà, notre petit tour des nouveautés du C99 est terminé ! Je n'ai fait que vous présenter les nouveautés les plus importantes et sans approfondir, volontairement, certains points. Par exemple les nombres flottants ou les caractères étendus nécessiteraient un article entier, si le coeur vous en dit : <http://club.developpez.com/redaction/> ;). Cependant à l'heure actuelle, il n'est pas recommandé d'utiliser les spécificités de cette nouvelle norme puisqu'elle n'est pas encore intégralement implémentée dans tous les compilateurs (par exemple pour gcc : [Status of C99 features in GCC](#)). Il existe tout de même certains compilateurs conformes à la norme C99 (6) :

- Dinkum Unabridged Library (Dinkumware, Ltd)
- EDG C/C++ Compiler (Edison Design Group)
- Visual Age Compiler (IBM)
- LMPCC C99 Compiler (Lund Multiprocessor Compiler Company, AB)
- Sun Studio 9 ( Sun Microsystems)



## VII - Références

Si vous souhaitez en savoir plus sur ce sujet, vous pouvez télécharger gratuitement la dernière version de la norme au format pdf : [n1124.pdf](#).

Je vous conseille aussi la lecture de deux très bons livres qui abordent le sujet : [méthodologie de la programmation en C](#) et [C précis et concis](#).

## VIII - Remerciements

Merci à **fearyourself**, **neguib** et à **Yogui** pour leur relecture attentive de cet article.

- 1 : **American National Standard Institut**
- 2 : **International Organization for Standardization**
- 3 : il s'agit d'une variable et non d'une macro comme `__FILE__` puisque le préprocesseur n'a pas la notion de fonction
- 4 : Cette dernière taille n'est définie que pour les processeurs 64 bits
- 5 : la taille est choisie afin d'augmenter la rapidité des opérations
- 6 : Certifié conforme par **Perennial**