Lucas-84
Taurre

Zeste de savoir

Aliasing et pointeurs restreints

15 janvier 2019

Table des matières

1. Introduction

Depuis ses débuts, le langage C pose un problème assez gênant aux compilateurs désireux d'optimiser le code, dû à son utilisation massive des pointeurs : le risque d'*aliasing* (ou « risque de chevauchement »).

Les normes successives ont tenté de l'atténuer à l'aide de la règle de strict *aliasing* (C89) et des pointeurs restreints (C99) ; deux concepts qui vont retenir notre attention dans ce tutoriel.

2. Une histoire d'aliasing

Dans un premier temps, nous allons découvrir cette notion d'*aliasing* et voir en quoi elle complique le travail du compilateur.

2.1. Rappels

2.1.0.1. La notion d'objet

> object : region of data storage in the execution environment, the contents of which can represent values.
>
> *ISO/IEC 9899 :2011, doc. N1570, § 3.15, Terms, definitions, and symbols, al. 1, p. 6.*

Le terme d'**objet**, qui sera au centre de nos discussions futures, désigne simplement une zone mémoire pouvant contenir des données.

2.1.1. La notion de lvalue

> An lvalue is an expression that [...] designates an object [...].
>
> *ISO/IEC 9899 :2011, doc. N1570, § 6.3.2.1, Lvalues, arrays, and function designators, al. 1, p. 54.*

Une **lvalue** est une expression qui désigne un objet (que ce soit pour un accès ou une modification).

```c
int i;
int j;
int *p;

/*
 * `p' est une lvalue car elle modifie un objet.
 * `&i' n'est pas une lvalue.
 */
p = &i;

/* `i' est une lvalue car elle modifie un objet. */
i = 10;

/*
 * `j' est une lvalue car elle modifie un objet.
 * `i' est une lvalue car elle accède à un objet.
 */
j = i;

/* `*p' est une lvalue car elle modifie un objet. */
*p = 30;
```

Gardez bien ces deux notions à l'esprit, nous allons en avoir besoin.

2.2. Présentation de la notion d'aliasing

En programmation, un cas d'**aliasing** se produit lorsque plusieurs *lvalues* désignent le même objet ; celles-ci sont alors qualifiées d'**alias**.

> *i* Certaines de ces situations dépassent le cadre du présent tutoriel. Pour notre part, nous nous intéresserons uniquement aux alias résultant de l'utilisation de pointeurs, car ce sont ceux qui engendrent les difficultés les plus importantes.

Par exemple, dans le code source ci-dessous, `*p` est un alias de `n`, c'est-à-dire que toute modification de `*p` aura une répercussion sur la valeur de `n` et *vice versa*.

```
1 int n;
2 int *p = &n;
```

Cette définition peut paraître simple, mais elle comporte aussi sa part de subtilités.

```
1 int a[2][10];
2
3 int *p = a[0];
4 int *q = a[1];
```

On serait ici tenté de dire que les *lvalues* `*p` et `*q` accèdent au même objet (le tableau `a`), pourtant il n'en est rien. En effet, il ne faut pas perdre de vue que la notion d'objet est étrangère à celle de type. Dès lors, un même objet peut être subdivisé en deux autres, indépendants l'un de l'autre. `*p` et `*q` ne sont donc pas des alias.

De la même manière, dans le code ci-dessous, `*p`, `*q`, `*r` et `*s` ne le sont pas non plus.

```
1 char a[4];
2
3 char *p = a + 0;
4 char *q = a + 1;
5 char *r = a + 2;
6 char *s = a + 3;
```

Cette division fonctionne jusqu'au plus petit objet possible (à savoir un *bit* dans le cas des champs de *bits*). Ainsi, `a.i` n'est pas un alias de `a.j`.

```
1 struct s {
2         unsigned int i : 1;
```

```
3         unsigned int j : 1;
4 };
5
6 struct s a;
```

2.3. Problématique d'optimisation du compilateur

Si les relations d'*aliasing* qui existent entre les différentes *lvalues* du programme ne sont pas préoccupantes pour le programmeur, cela l'est plus pour le compilateur, qui peut être gêné dans son travail d'optimisation.

2.3.1. Problèmes causés par les alias au compilateur

Après avoir vérifié que le code source est syntaxiquement correct, le compilateur entre dans une seconde phase : celle de l'**optimisation de code**. Cette étape consiste simplement en la modification du code dans le but que l'exécution du programme se déroule le plus rapidement possible. Pour cela, il va prendre en compte certains éléments de l'implémentation, comme les différentes instructions dont dispose le processeur. Pour le moment, nous nous concentrerons uniquement sur une des optimisations les plus basiques : la **réorganisation du code**.

```
1 ##include <stdio.h>
2
3 void
4 f(void)
5 {
6         const int n = 5;
7
8         printf("%d\n", n);
9 }
```

Ici, force est de constater que l'instruction de la ligne 6 est inutile, puisque la variable `n` n'est pas modifiée. Aussi le compilateur pourra-t-il, par exemple, remplacer le code d'appel de cette fonction par un simple `printf("%d ", 5)`.

Le problème dans tout cela, c'est que l'aliasing complique cette réorganisation. En effet, dans le code ci-dessous, on peut se dire à première vue que le compilateur pourrait supprimer la ligne 8 et remplacer l'instruction de la ligne 10. Or, si les lvalues `*p` et `n` sont des alias, le résultat attendu est complètement différent (10 en l'occurrence). Par conséquent, compte tenu du risque d'aliasing, le compilateur est obligé de laisser ces instructions telles quelles (ce qui peut, à long terme, ralentir l'exécution du programme).

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

static int n;

void
f(int *p)
{
        n = 5;
        *p = 10;
        printf("%d\n", n);
}
```

2.3.2. Analyse d'alias par le compilateur

L'**analyse d'alias**, c'est-à-dire la recherche des situations d'*aliasing* dans un programme donné, est donc nécessaire pour le compilateur, afin de pouvoir déterminer quels cas peuvent permettre telle ou telle optimisation.

Peu importe le résultat de cette analyse (alias ou pas) : dans les deux cas, une optimisation pourra être effectuée. La seule situation problématique se produit lorsqu'on ne peut pas déterminer, lors de la compilation, les relations d'*aliasing* qui existent entre deux *lvalues*. Le compilateur est alors obligé de considérer le pire des cas : on parle d'**aliasing pessimiste**.

```c
*p = 4;
*q = 6;
n = *p + *q;
```

Avec uniquement ces informations, le compilateur se retrouve face à trois cas distincts (en supposant que `*p`, `*q` et `n` sont de type `int`) :

- si `*p` et `*q` ne sont pas des alias, alors `n = *p + *q` pourra être remplacé par `n = 10` ;
- si `*p` et `*q` sont des alias, alors `n = *p + *q` pourra être remplacé par `n = 12` ;
- si il est impossible de déterminer les relations d'*aliasing* qui existent entre `*p` et `*q`, alors le code ne pourra pas être modifié.

Le compilateur se doit donc d'effectuer une analyse d'alias pertinente pour sélectionner une de ces affirmations (et, si possible, une des deux premières). Dans cette optique, beaucoup algorithmes ont été développés. Néanmoins, en pratique, la plupart sont trop lourds pour être intégrés aux compilateurs courants, si bien que ces derniers se contentent généralement d'une analyse superficielle (ce qui peut se révéler pénalisant pour les performances).

3. La règle de strict aliasing

Nous avons donc vu en quoi il était important pour le travail d'optimisation du compilateur de connaître les relations d'*aliasing* qui existent entre les *lvalues* du programme. Cette préoccupation a été au centre de beaucoup de critiques du langage C à ses débuts, qui lui reprochaient son imprécision dans l'analyse des pointeurs. Aussi la norme aide-t-elle l'analyse d'alias avec un premier concept : la **règle de strict aliasing**.

3.1. Normalisation de la règle de strict aliasing

Nous allons maintenant faire un petit tour d'horizon de la définition et de la normalisation de cette règle en suivant un ordre chronologique (de son inauguration dans la norme C89 à sa précision dans la norme C99).

3.1.1. La norme C89

C'est en 1989 que le comité de l'ANSI décida de l'instaurer, dans le but de réduire le nombre de cas d'*aliasing* pessimistes. Grâce à cela, le compilateur a pu affiner son analyse d'alias en présumant des *lvalues* comme n'étant pas des alias en fonction de leur type et de celui de l'objet qu'elles désignent.

Un objet n'a techniquement pas de type (ce n'est qu'une zone mémoire pouvant contenir des données). Cependant, afin de faciliter l'analyse d'alias, la norme leur en a fixé fictivement un.

Voyons maintenant l'énoncé de la règle.

> An object shall have its stored value accessed only by an lvalue that has one of the following types :
> — the declared type of the object ;
> — a qualified version of the declared type of the object ;
> — a type that is the signed or unsigned type corresponding to the declared type of the object ;
> — a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object ;
> — an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union) ; or
> — a character type.
>
> *C89 (X3J11/88-090), § 3.3, Expressions, al. 6.*

Un objet ne peut être accédé que par une *lvalue* qui a un des types suivants :

— un type identique au type déclaré de l'objet ;
— une version qualifiée du type déclaré de l'objet ;
— un type qui est le type signé ou non signé correspondant au type déclaré de l'objet ;

— un type qui est le type signé ou non signé correspondant à une version qualifiée du type déclaré de l'objet ;
— un agrégat ou une union qui inclut un des types mentionnés ci-dessus parmi ses membres (incluant, de manière récursive, les sous-agrégats ou les sous-unions) ;
— un type caractère.

La norme se base sur le **type déclaré** de l'objet, qui lui est attribué lors de sa définition. Par exemple, dans le code ci-dessous, deux objets sont créés, ayant respectivement comme type déclaré le type `int` et le type `double`.

```
1 int n;
2 double x;
```

3.1.2. La norme C99

La norme C99 a peaufiné cette règle en la rapportant, non plus au type déclaré, mais au **type effectif** de l'objet, une nouvelle notion qui permet de mieux gérer le cas dans lequel l'objet ne dispose pas d'un type déclaré. Cela vise essentiellement les objets alloués dynamiquement, puisque ces derniers ne sont pas créés lors d'une définition, mais lors d'un appel à une fonction d'allocation.

> The effective type of an object for an access to its stored value is the declared type of the object, if any. If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using memcpy or memmove, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access. Source : ISO/IEC 9899 :TC3, doc. N1256, § 6.5, Expressions, al. 6, pp. 67-68.

Dans le cas où un objet n'a pas de type déclaré, son type effectif est :

— celui de la *lvalue* le désignant (accès ou modification) ;
— celui de l'objet dont le contenu y a été copié à l'aide de `memcpy` ou `memmove`.

```
1 ##include <stdlib.h>
2
3 int *p = malloc(sizeof *p);
4
5 /*
6  * Le type de l'objet désigné par la lvalue `*p' prend le type
7  * `int'.
8  */
9 *p = 10;
```

```
10
11 /*
12  * Le type de l'objet désigné par la lvalue `*p' prend le type
13  * `unsigned int'.
14  */
15 *(unsigned int *)p = 20U;
```

Pour conclure, notons que la norme C11 n'a pas changé l'énoncé de la règle.

3.2. Illustrations de la règle de strict aliasing

Vous devriez maintenant être au point avec la définition et la normalisation de la règle de strict *aliasing*. Pour illustrer un peu nos propos, nous étudierons tout d'abord quelques exemples et contre-exemples, puis nous verrons quel intérêt le compilateur peut tirer de tout cela.

3.2.1. Exemples

La question sera de savoir, pour chacune des lignes de code ci-dessous, si l'utilisation de l'alias créé est autorisé.

```
1  unsigned int n;
2
3  /*
4   * Correct, car la lvalue `*p' a un type qui est une version
5   * qualifiée du type effectif de l'objet.
6   */
7  const unsigned int *p = &n;
8
9  /*
10  * Incorrect, car la lvalue `*q' a un type qui n'est ni une version
11  * qualifiée du type déclaré de l'objet, ni le type signé ou non
12  * signé correspondant, ni un type caractère.
13  */
14 long int *q = (long int *)&n;
15
16 /*
17  * Correct, car la lvalue `*r' a un type qui est le type signé
18  * correspondant à une version qualifiée du type effectif de
        l'objet.
19  */
20 const int *r = &n;
21
22 /* Correct, car la lvalue `*s' a un type caractère. */
23 signed char *s = (signed char *)&n;
```

Mentionnons que toutes ces règles s'appliquent uniquement lors du déférencement, ce qui autorise donc en soi l'affectation (bien que, naturellement, le champ d'action du pointeur soit par la suite réduit puisqu'il sera interdit de le déférencer).

```
1 int n;
2
3 /* Correct : `p' n'est pas déférencé. */
4 short int *p = (short int *)&n;
5
6 /* Incorrect : `p' est déférencé. */
7 *p = 10;
```

3.2.2. Bénéfices pour le compilateur

Pour le compilateur, le plus intéressant reste la conséquence de cette règle, c'est-à-dire qu'en dehors des accès autorisés mentionnés ci-dessus, deux *lvalues* ne désigneront jamais un même objet.

Dans l'exemple ci-dessus, la règle de strict *aliasing* est brisée car `*p` a un type qui n'est ni une version qualifiée du type déclaré de l'objet, ni le type signé ou non signé correspondant, ni un type caractère. Les *lvalues* `*p` et `n` ne seront donc pas considérées comme des alias lors de la phase d'optimisation (bien qu'en vérité elles le soient). Voilà qui facilite bien l'analyse d'alias !

3.2.3. Paramétrage du compilateur gcc

Avec le compilateur gcc, la règle de strict *aliasing* n'est activée par défaut que dans les niveaux d'optimisation. Toutefois, il est possible pour le programmeur de spécifier explicitement s'il veut que son code subisse les vérifications associées, à l'aide des options `-fstrict-aliasing` (respect strict de la règle) et `-fno-strict-aliasing` (tolérance de comportements non conformes à la règle).

Si l'utilisation pertinente de l'option `-fno-strict-aliasing` peut vous paraître dangereuse, puisque le code n'est alors plus conforme à la norme, l'histoire retient que de grands noms l'ont soutenu (le noyau Linux pour ne citer que lui).

gcc dispose notamment d'un avertissement permettant de prévenir les situation non conformes à la règle de strict *aliasing*.

> warning : dereferencing type-punned pointer will break strict-aliasing rules

L'option permettant de gérer de tels affichages est `-Wstrict-aliasing[=n]`, avec `n` compris entre 1 et 3 (niveau 3 par défaut). Plus `n` est petit, moins gcc fera de vérifications (par exemple, avec `n = 1` ou `n = 2`, l'avertissement peut se déclencher même si le pointeur n'est pas déférencé). De même, le pourcentage de faux positifs et de faux négatifs dépend du niveau utilisé.

Par exemple, avec gcc 4.9.2, l'avertissement ne se déclare qu'aux niveaux 1 et 2 pour ce code, au niveau de l'instruction d'affectation (ligne 5).

```c
int
main(void)
{
        unsigned int n;
        long int *p = (long int *)&n;

        *p = 10L;
        return 0;
}
```

Si la règle de strict *aliasing* constitue une aide réelle à l'analyse d'alias des compilateurs, il reste encore le cas des alias de type identique (ou ne différant que par le signe et/ou par le qualificateur, ainsi que celui des types caractères). Il est évident qu'on ne peut pas interdire cette pratique, qui signifierait l'abolition des pointeurs ! Mais c'est à ce moment-là que le programmeur entre en scène avec l'*aliasing* spécifié, thème qui fera l'objet de notre prochaine sous-partie.

4. Les pointeurs restreints

Malgré cette règle, il reste donc encore quelques situations d'*aliasing* compromettantes. Comme la norme et les compilateurs ne peuvent plus faire d'hypothèses supplémentaires, c'est le programmeur lui-même qui est sollicité.

4.1. Introduction aux pointeurs restreints

L'*aliasing* spécifié par le développeur est implémenté dans la norme C99 sous la forme de la notion de **pointeur restreint**. L'idée est de permettre la mise en place d'un droit exclusif d'accès sur un objet référencé par un pointeur qualifié de restreint. Ce droit ne peut être transmis qu'à des *lvalues* dérivées de ce pointeur, c'est-à-dire qui ont obtenu l'adresse de l'objet *via* celui-ci.

4.2. Le qualificateur restrict

Pour déclarer un pointeur restreint, la norme C99 a mis à la disposition des programmeurs un nouveau qualificateur : `restrict`. Il est applicable uniquement aux pointeurs sur objet ; il doit donc être placé, lors de la déclaration, après le symbole `*`.

```c
/* `p' est un pointeur restreint sur `int'. */
int * restrict p;

/* `q' est un pointeur sur pointeur restreint sur `int'. */
int * restrict *q;

```

```
/* `r' est un pointeur restreint sur pointeur sur `int'. */
int ** restrict r;
```

4.2.1. Définition

Le passage à propos de `restrict` dans la norme C99 peut paraître alambiqué et tordu ; nous vous ferons donc grâce des citations. L'essentiel du fonctionnement des pointeurs restreints peut se résumer dans les deux règles suivantes.

1. il ne peut y avoir qu'un seul pointeur restreint référençant un même objet dans un même bloc ;
2. une *lvalue* ne peut modifier un objet référencé par un pointeur restreint que si elle est dérivée de ce dernier.

4.2.2. Quelques exemples

À ce stade, la définition peut vous paraître encore un peu floue, c'est pourquoi nous vous proposons quelques petits exemples.

```
##include <stdlib.h>

static char * restrict r, * restrict s;

void
copy(void * restrict dst, void * restrict src, size_t n)
{
        /*
         * Valide car `p' et `q' ne sont pas des pointeurs
         * restreints.
         */
        char *p = dst;
        char *q = src;

        while (n-- != 0) {
                /*
                 * Valide car les lvalues `*p' et `*q' sont
                 * respectivement basées sur les pointeurs
    restreints
                 * `dst' et `src'.
                 */
                *p++ = *q++;
        }

        /*
         * Invalide car `r' et `s' sont des pointeurs restreints et
```

```
26          * référencent, respectivement, les mêmes objets que `dst'
    et
27          * `src' dans le bloc de la fonction `copy'.
28          */
29         *r = *s;
30 }
31
32 int
33 main(void)
34 {
35         /*
36          * Valide car `r' et `s' référencent deux objets différents
37          * dans le bloc de la fonction main.
38          */
39         r = malloc(10);
40         s = malloc(10);
41
42         /*
43          * Valide car `dst' et `src' référencent deux objets
44          * différents dans le bloc de la fonction `copy'.
45          */
46         copy(r, s, 10);
47
48         /*
49          * Invalide car `dst' et `src' référencent le même objet
    dans
50          * le bloc de la fonction `copy'.
51          */
52         copy(r, r, 10);
53
54         /*
55          * Invalide car `r' et `s' référencent alors le même objet
56          * dans le bloc de la fonction main.
57          */
58         r = s;
59         return 0;
60 }
```

Bien qu'il soit techniquement possible d'assigner des pointeurs restreints à des pointeurs non restreints, c'est une pratique déconseillée car cela peut compliquer le travail d'optimisation du compilateur.

4.3. Bénéfice des pointeurs restreints

À l'instar de la règle de strict *aliasing*, les pointeurs restreints permettent aux compilateurs d'effectuer des présomptions quant aux relations d'*aliasing* qui existent entre les pointeurs d'un même bloc. En effet, deux pointeurs restreints sont garantis de ne pas être des alias. Ainsi, c'est toute la phase d'optimisation de code qui en profite, et notamment la réorganisation du code.

4.3.1. La vectorisation

De plus, étant donné que le mot-clé `restrict` vise des pointeurs de même type, cela laisse également place à une optimisation plus poussée faisant intervenir les tableaux : la **vectorisation**. Cette dernière pratique consiste à effectuer des opérations sur des petits tableaux de taille fixe plutôt que sur un seul élément à la fois. Cela est possible sur la plupart des processeurs modernes, qui disposent d'instructions spécialisées travaillant sur plusieurs éléments à la fois : les **instructions vectorielles**.

Dans les exemples suivants, nous considérerons un processeur disposant d'instructions vectorielles capables de travailler sur 128 *bits*, c'est-à-dire ici de 16 `char` ou de 4 `int`. Elles seront illustrées par les trois fonctions suivantes :

— `vect_cpy16` : copie un tableau de 16 `char` ;
— `vect_cpy4` : copie un tableau de 4 `int` ;
— `vect_add4` : additionne deux tableaux de 4 `int` et stocke le résultat dans le premier.

4.3.2. Exemple (1)

```
1  void
2  memcpy(void * restrict dst, void * restrict src, size_t n)
3  {
4          char *p = dst;
5          char *q = src;
6
7          while (n-- != 0)
8                  *p++ = *q++;
9  }
```

Dans le code ci-dessus, la règle de strict *aliasing* est inutile car les *lvalues* `*p` et `*q` sont toutes deux de type `char`. En revanche, elles sont basées sur un pointeur restreint et sont donc garanties de ne pas être des alias. Le compilateur pourrait donc vectoriser cette boucle, en copiant des tableaux de taille fixe plutôt que des `char` un par un.

```
1  void
2  memcpy(void * restrict dst, void * restrict src, size_t n)
3  {
4          char *p = dst;
5          char *q = src;
6          size_t i;
7
8          for (i = 0; n - i >= 16; i += 16)
9                  vect_cpy16(p + i, q + i);
10
11         for (; i < n; ++i)
12                 p[i] = q[i];
```

```
13 }
```

4.3.3. Exemple (2)

```
1 void
2 vect_add(int * restrict res, int * restrict a, int * restrict b,
      size_t n)
3 {
4         for (size_t i = 0; i < n; ++i)
5                 res[i] = a[i] + b[i];
6 }
```

De la même manière, le code ci-dessus opère sur des *lvalues* basées sur des pointeurs restreints. Ainsi, le compilateur pourrait utiliser des instructions vectorielles afin d'optimiser le code comme suit.

```
1  void
2  vect_add(int * restrict res, int * restrict a, int * restrict b,
       size_t n)
3  {
4          size_t i;
5
6          for (i = 0; n - i >= 4; i += 4) {
7                  vect_cpy4(res + i, a + i);
8                  vect_add4(res + i, b + i);
9          }
10
11         for (; i < n; ++i)
12                 res[i] = a[i] + b[i];
13 }
```

4.4. Dangers des pointeurs restreints

Malgré tout, il est important de prendre des précautions lors de l'utilisation des pointeurs restreints ; ils ne doivent en effet pas être utilisés à tout-va.

4.4.1. Confusion entre appelant et appelé

Si deux pointeurs sont indiqués comme étant restreints mais, qu'en réalité, ils se chevauchent, le résultat est indéterminé et le code produit a de fortes chances d'être incorrect.

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

void
f(void)
{
        int a[8] = { 0, 0, 45, 42, 12, 89, 2, 36 };

        /*
         * Les deux arguments restreints de `memcpy' se chevauchent,
         * c'est une situation de comportement indéterminé.
         */
        memcpy(a, a + 2, 6);
}
```

Or, nous pouvons remarquer que c'est à la fonction appelée de préciser si les arguments doivent être restreints, mais seule la fonction appelante peut contrôler si ces arguments sont conformes (le compilateur ne peut pas faire cette vérification par lui-même).

4.4.2. Précautions d'utilisation

On distingue deux grands cas dans lesquels on peut utiliser les pointeurs restreints.

1. Si l'algorithme de la fonction ne fonctionne pas ou n'a aucun sens dans le cas où les paramètres se chevauchent, alors le résultat avec les pointeurs restreints sera toujours incorrect, mais pourra avoir changé. Par exemple, le chevauchement des deux paramètres dans la fonction `fopen` serait complètement absurde.
2. Si le principe de la fonction a un sens dans le cas où les arguments se chevauchent mais que cela est pénalisant pour les optimisations, alors il est préférable de créer deux versions de la fonction (une avec `restrict` et une sans), à la manière des fonctions `memcpy` et `memmove`.

Lors d'un comportement indéterminé, théoriquement, tout peut se passer. Le compilateur a donc tout à fait le droit de produire un code qui arrête brutalement le programme pour éviter de propager une éventuelle erreur. C'est pourquoi il ne faut pas oublier de prévenir l'utilisateur de ces spécifications dans la documentation de la fonction. C'est par exemple le cas pour la fonction `memcpy`.

Au final, on peut représenter tout cela par une sorte de contrat passé entre les deux fonctions. Si il n'est pas respecté par la fonction appelante, alors la fonction appelée se réserve le droit de produire un code incorrect.

4.5. Une optimisation vraiment valable ?

Il ne faut pas oublier que `restrict` est une simple indication et pas une obligation pour le compilateur. Aussi les détracteurs du mot-clé ont-ils souvent souligné le fait que le gain de temps octroyé par l'utilisation des pointeurs restreints n'est pas toujours très important (par exemple, les processeurs qui ne disposent pas d'instructions vectorielles ne jouiront pas de cette optimisation).

Nous pouvons donc légitimement nous demander si l'utilisation des pointeurs restreints est réellement rentable par rapport à l'effort de réflexion associé (qui est loin d'être négligeable). Plusieurs travaux ont conclu que ce n'était pas le cas, et déconseillent donc l'*aliasing* spécifié.

À vous de vous forger votre propre avis ; n'hésitez pas, dans cette optique, à construire vos propres étalonnages suivant votre utilisation du langage. En tout cas, si vous êtes prêts à fournir un effort supplémentaire pour un bénéfice, si minimal qu'il soit, vous voilà informés !

5. Conclusion

Ainsi, ce tutoriel touche à sa fin. Nous espérons vous avoir éclairé sur ce difficile sujet d'aliasing et de pointeurs restreints. La norme du langage C est, aujourd'hui encore, une des seules normes de langage de programmation qui prône l'optimisation des compilateurs, le C cherche donc toujours à prouver ses qualités en performances pures.

6. Liens externes

6.1. Alias et optimisation

— (en) Optimisation de code par le compilateur sur Wikipédia .
— (en) Optimisation de boucles sur Wikipédia .
— (fr) Pipeline des instructions sur Wikipédia .
— (en) Les processeurs vectoriels sur Wikipédia .
— (en) Guide de programmation pour la vectorisation des compilateurs C/C++ .

6.2. Analyse d'alias

— (en) Analyse d'alias sur Wikipédia .
— (en) Exemple d'algorithme effectuant une analyse d'alias performante .
— (en) L'analyse d'alias de gcc .

6.3. Règle de strict aliasing

— (en) Comprendre le strict aliasing .
— (en) Type-punning et strict aliasing .
— (en) Comprendre le strict aliasing en C/C++ .

6.4. Paramétrage du compilateur

— (en) Voir les raisons de l'utilisation de -fno-strict-aliasing dans le noyau Linux .
— (en) Page de manuel de gcc (recherchez « -fstrict-aliasing ») .

6.5. Les pointeurs restreints

— (fr) Le C et ses raisons : les pointeurs restreints .
— (en) Les pointeurs restreints en C .
— (en) Defect Report #294 .
— (en) Démystification du mot-clé restrict .
— (en) Pourquoi l'*aliasing* spécifié est une mauvaise idée .